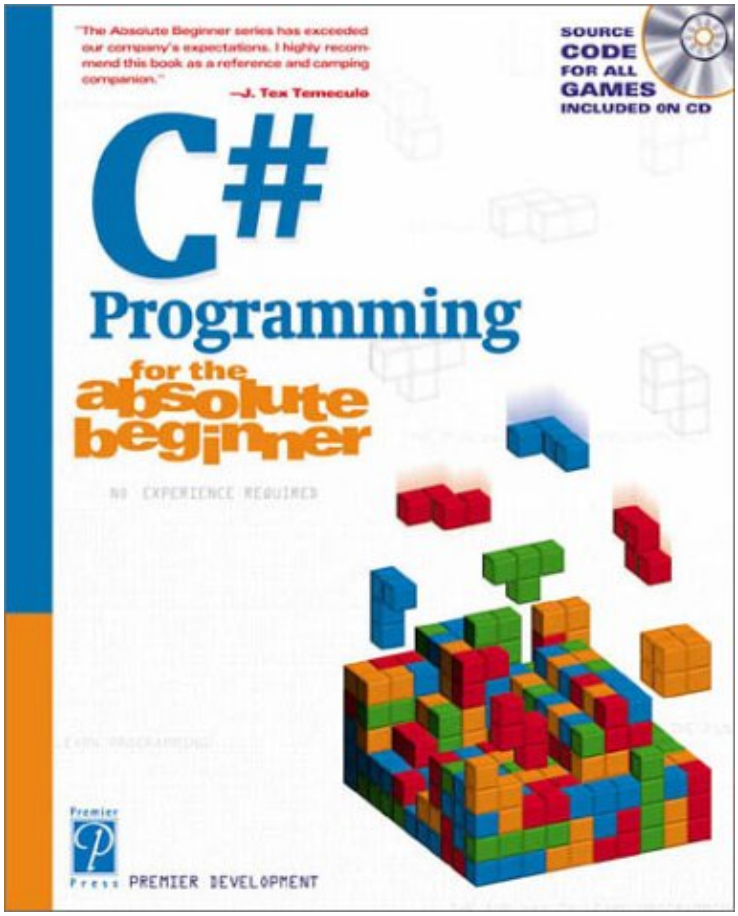
VISIT…


"The Absolute Beginner series has exceeded our company's expectations. I highly recommend this book as a reference and camping companion."
—J. Tex Temecula
SOURCE CODE FOR ALL GAMES INCLUDED ON CD
C#
Programming
for the absolute beginner
NO EXPERIENCE REQUIRED
Premier Press
PREMIER DEVELOPMENT

Microsoft C# Programming for the Absolute Beginner

Table of Contents

Microsoft C# Programming for the Absolute Beginner....1

Introduction....4
Overview....4

Chapter 1: Basic Input and Output: A Mini Adventure....5
Project: The Mini Adventure....5
Reviewing Basic C# Concepts....6
Namespaces....7
Classes....7
Methods....7
Statements....7
The Console Object....8
.NET Documentation....8
Saying “Hello, World!”....12
Getting into the Visual Studio .Net Environment....13
Examining the Default Code....16
Creating a Custom Namespace....16
Adding Summary Comments....17
Creating the Class....17
Moving from Code to a Program....19
Compiling Your Program....20
Looking for Bugs....21
Getting Input from the User....22
Creating a String Variable....24
Getting a Value with the Console.ReadLine() Method....24
Incorporating a Variable in Output....25
Combining String Values....26
Combining Strings with Concatenation....27
Adding a Tab Character....27
Using the Newline Sequence....27
Displaying a Backslash....27
Displaying Quotation Marks....27
Launching the Mini Adventure....28
Planning the Story....28
Creating the Variables....28
Getting Values from the User....29
Writing the Output....29
Finishing the Program....30
Summary....31

Chapter 2: Branching and Operators: The Math Game....32
The Math Game....32
Using Numeric Variables....33
The Simple Math Game....33
Numeric Variable Types....34
Integer Variables....35
Long Integers....36
Floating−Point Variables....36
Data Type Problems....37

Table of Contents

Chapter 2: Branching and Operators: The Math Game
Math Operators....37
Converting Variables....37
Explicit Casting....39
The Convert Object....39
Creating a Branch in Program Logic....41
The Hi Bill Game....41
Condition Testing....43
The If Statement....44
The Else Clause....44
Multiple Conditions....44
Working with The Switch Statement....45
The Switch Demo Program....45
Examining How Switch Statements Work....46
Creating a Random Number....47
Introducing the Die Roller....47
Exploring the Random Object....48
Creating a Random Double with the .NextDouble() Method....48
Getting the Values of Dice....49
Creating the Math Game....50
Designing the Game....50
Creating the Variables....50
Managing Addition....51
Managing Subtraction....52
Managing Multiplication and Division....52
Checking the Answers....53
Waiting for the Carriage Return....53
Summary....54

Chapter 3: Loops and Strings: The Pig Latin Program....55
Project: The Pig Latin Program....55
Investigating The String Object....56
The String Mangler Program....56
A Closer Look at Strings....56
Using the Object Browser....57
Experimenting with String Methods....58
Performing Common String Manipulations....59
Using a For Loop....60
Examining The Bean Counter Program....60
Creating a Sentry Variable....61
Checking for an Upper Limit....61
Incrementing the Variable....61
Examining the Behavior of the For Loop....61
Varying the For Loop's Behavior....62
The Fancy Beans Program....63
Skipping Numbers....64
Counting Backwards....64
Using a Foreach Loop to Break Up a Sentence....65
Using a While Loop....65
The Magic Word Program....66

Table of Contents

Chapter 3: Loops and Strings: The Pig Latin Program
Writing an Effective While Loop....68
Planning Your Program with the STAIR Process....70
S: State the Problem....70
T: Tool Identification....70
A: Algorithm....71
I: Implementation....71
R: Refinement....72
Applying STAIR to the Pig Latin Program....72
Stating the Problem....73
Identifying the Tools....73
Creating the Algorithm....73
Implementing and Refining....74
Writing the Pig Latin Program....74
Setting Up the Variables....74
Creating the Outside Loop....75
Dividing the Phrase into Words....75
Extracting the First Character....76
Checking for a Vowel....76
Adding Debugging Code....76
Closing Up the code....77
Summary....77

Chapter 4: Objects and Encapsulation: The Critter Program....78
Introducing the Critter Program....78
Creating Methods to Reuse Code....80
The Song Program....80
Building the Main() Method....81
Creating a Simple Method....82
Adding a Parameter....83
Returning a Value....84
Creating a Menu....85
Creating a Main Loop....85
Creating the Sentry Variable....86
Calling a Method....86
Working with the Results....87
Writing the showMenu() Method....87
Getting Input from the User....87
Handling Exceptions....88
Returning a Value....89
Creating a New Object with the CritterName Program....89
Creating the Basic Critter....89
Using Scope Modifiers....90
Using a Public Instance Variable....90
Creating an Instance of the Critter....91
Referring to the Critter's Members....91
Adding a Method....91
Creating the talk() Method for the CritterTalk Program....92
Changing the Menu to Use the talk() Method....92
Creating a Property in the CritterProp Program....92

Table of Contents

Chapter 4: Objects and Encapsulation: The Critter Program
Examining the Critter Prop Program....93
Creating the Critter with a Name Property....93
Using Properties as Filters....95
Making the Critter More Lifelike....96
Adding More Private Variables....96
Adding the Age() Method....97
Adding the Eat() Method....97
Adding the Play() Method....98
Modifying the Talk() Method....98
Making Changes in the Main Class....98
Summary....99

Chapter 5: Constructors, Inheritance, and Polymorphism: The Snowball Fight....101
Introducing the Snowball Fight....101
Inheritance and Encapsulation....102
Creating a Constructor....102
Adding a Constructor to the Critter Class....103
Creating the CritViewer Class....104
Reviewing the Static Keyword....105
Calling a Constructor from the Main() Method....106
Examining CritViewer's Constructor....106
Working with Multiple Files....107
Overloading Constructors....108
Viewing the Improved Critter Class....108
Adding Polymorphism to Your Objects....109
Modifying the Critter Viewer in CritOver to Demonstrate Overloaded Constructors....110
Using Inheritance to Make New Classes....111
Creating a Class to View the Clone....112
Creating the Critter Class....113
Improving an Existing Class....113
Introducing the Glitter Critter....114
Calling the Base Class's Constructors....115
Adding Methods to a New Class....116
Changing the Critter Viewer Again....116
Using Polymorphism to Alter a Class's Behavior....116
Creating the Snowball Fight....117
Building the Fighter....118
Building the Robot Fighter....120
Creating the Main Menu Class....122
Summary....125

Chapter 6: Creating a Windows Program: The Visual Critter....127
Overview....127
Introducing the Visual Critter....127
Creating a Windows–Style Program with a GUI....134
Thinking Like a GUI Programmer....135
Creating a Graphical User Interface (GUI)....136
Examining the Code of a Windows Program....139
Adding New Namespaces....140

Table of Contents

Chapter 6: Creating a Windows Program: The Visual Critter
Creating the Form Object....142
Creating a Destructor....143
Creating the Components....144
Setting Component Properties....145
Setting Up the Form....145
Writing the Main() Method....146
Creating an Interactive Program....147
Responding to a Simple Event....147
Creating and Adding the Components....148
Adding an Event to the Program....148
Creating an Event Handler....149
Allowing for Multiple Selections....150
Choosing a Font with Selection Controls....150
Creating the User Interface....151
Examining Selection Tools....153
Creating Instance Variables in the Font Chooser....154
Writing the AssignFont() Method....155
Writing the Event Handlers....157
Working with Images and Scroll Bars....157
Changing an Image's Size....158
Setting Up the Picture Box....159
Adding a Scroll Bar....161
Writing the Event–Handling Code....161
Revisiting the Visual Critter....161
Designing the Program....162
Determining the Necessary Tools....163
Designing the Form....163
Writing the Code....164
Summary....167

Chapter 7: Timers and Animation: The Lunar Lander....168
Introducing the Lunar Lander....168
Reading Values from the Keyboard....169
Introducing the Key Reader Program....169
Setting Up the Key Reader Program....171
Coding the KeyPress Event....171
Coding the KeyDown Event....173
Determining Which Key Was Pressed....175
Animating Images....175
Introducing the ImageList Control....176
Setting Up an Image List....177
Looking at the Image Collection....178
Displaying an Image from the Image List....179
Using a Timer to Automate Animation....180
Introducing the Timer Control....180
Configuring the Timer....181
Adding Motion....182
Checking for Keyboard Input....184
Working with the Location Property....184

Table of Contents

Chapter 7: Timers and Animation: The Lunar Lander
Detecting Collisions between Objects....186
Coding the Crasher Program....188
Getting Values for newX and newY....189
Bouncing the Ball off the Sides....189
Checking for Collisions....189
Extracting a Rectangle from a Component....189
Getting More from the MessageBox Object....190
Introducing the MsgDemo Program....190
Retrieving Values from the MessageBox....192
Coding the Lunar Lander....192
The Visual Design....192
The Designer−Generated Code....193
Class−Level Variables....194
The Constructor....195
The timer1_Tick() Method....195
The moveShip() Method....196
The checkLanding() Method....197
The theForm_KeyDown() Method....199
The showStats() Method....200
The killShip() Method....200
The initGame() Method....201
Summary....202

Chapter 8: Arrays: The Soccer Game....203
The Soccer Game....203
Introducing Arrays....204
Exploring the Counter Program....205
Creating an Array of Strings....207
Referring to Elements in an Array....208
Working with Arrays....208
Using the Array Demo Program to Explore Arrays....208
Building the Languages Array....209
Sorting the Array....209
Creating Tables with Two−Dimensional Arrays....214
Designing the Soccer Game....214
Solving a Subset of the Problem....215
Adding Percentages for the Other Players....216
Setting Up the Shot Demo Program....216
Setting Up the List Boxes....217
Using a Custom Event Handler....218
Writing the changeStatus() Method....219
Kicking the Ball....219
Designing Programs by Hand....220
Examining the Form by Hand Program....220
Adding Components in the Constructor....221
Responding to the Button Event....222
Building the Soccer Program....222
Setting Up the Variables....222
Examining the Constructor....225

Table of Contents

Chapter 8: Arrays: The Soccer Game
Setting Up the Players....225
Setting Up the Opponents....227
Setting Up the Goalies....228
Responding to Player Clicks....228
Handling Good Shots....229
Handling Bad Shots....230
Setting a New Current Player....230
Handling the Passage of Time....231
Updating the Score....234
Summary....235

Chapter 9: File Handling: The Adventure Kit....236
Introducing the Adventure Kit....236
Viewing the Main Screen....236
Loading an Adventure....237
Playing an Adventure....238
Creating an Adventure....240
Reading and Writing Text Files....241
Exploring the File IO Program....242
Importing the IO Namespace....242
Writing to a Stream....243
Reading from a Stream....244
Creating Menus....245
Exploring the Menu Demo Program....245
Adding a MainMenu Object....246
Adding a Submenu....247
Setting Up the Properties of Menu Items....248
Writing Event Code for Menus....249
Using Dialog Boxes to Enhance Your Programs....250
Exploring the Dialog Demo Program....250
Adding Standard Dialogs to Your Form....253
Using the File Dialog Controls....253
Responding to File Dialog Events....254
Using the Font Dialog Control....255
Using the Color Dialog Control....256
Storing Entire Objects with Serialization....256
Exploring the Serialization Demo Program....256
Creating the Contact Class....257
Referencing the Serializable Namespace....258
Storing a Class....258
Retrieving a Class....259
Returning to the Adventure Kit Program....259
Examining the Room Class....260
Creating the Dungeon Class....263
Writing the Game Class....264
Writing the Editor Class....269
Writing the MainForm Class....274
Summary....276

Table of Contents

Chapter 10: Chapter Basic XML: The Quiz Maker....277
Introducing the Quiz Maker Game....277
Taking a Quiz....277
Creating and Editing Quizzes....278
Investigating XML....278
Defining XML....279
Creating an XML Document in .NET....283
Creating an XML Schema for Your Language....284
Investigating the .NET View of XML....285
Exploring the XmlNode Class....285
Exploring the XmlDocument Class....286
Reading an Existing XML Document....287
Creating the XML Viewer Program....293
Writing New Values to an XML Document....298
Designating the Class−Level Variables....298
Building the Document Structure....299
Adding an Element to the Document....300
Displaying the XML Code....301
Examining the Quizzer Program....302
Building the Main Form....303
Writing the Quiz Form....304
Writing the Editor Form....310
Summary....317

Chapter 11: Databases and ADO.NET: The Spy Database....318
Overview....318
Introducing the SpyMaster Program....318
Creating a Simple Database....321
Accessing the Data Server....321
Accessing the Data in a Program....326
Using Queries to Modify Data Results....333
Limiting Data with the SELECT Statement....333
Using an Existing Database....338
Adding the Capability to Display Queries....339
Creating a Visual Query Builder....340
Working with Relational Databases....345
Improving Your Data with Normalization....346
Using a Join to Connect Two Tables....347
Creating a View....350
Referring to a View in a Program....353
Incorporating the Agent Specialty Attribute....353
Working with Other Databases....355
Creating a New Connection....355
Converting a Data Set to XML....359
Reading from XML to a Data Source....360
Creating the SpyMaster Database....361
Building the Main Form....361
Editing the Assignments....362
Editing the Specialties....363
Viewing the Agents....364

Table of Contents

Chapter 11: Databases and ADO.NET: The Spy Database
Editing the Agent Data....365
Summary....374

List of Figures....375

List of Tables....382

List of Sidebars....383

Microsoft C# Programming for the Absolute Beginner

Andy Harris

© 2002 by Premier Press. All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system without written permission from Premier Press, except for the inclusion of brief quotations in a review.

The Premier Press logo, top edge printing, and related trade dress is a trademark of Premier Press, Inc. and may not be used without written permission. All other trademarks are the property of their respective owners.

Microsoft, Windows, Internet Explorer, Notepad, VBScript, ActiveX, and FrontPage are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

All other trademarks are the property of their respective owners.

Important: Premier Press cannot provide software support. Please contact the appropriate software manufacturer's technical support line or Web site for assistance.

Premier Press and the author have attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer.

Information contained in this book has been obtained by Premier Press from sources believed to be reliable. However, because of the possibility of human or mechanical error by our sources, Premier Press, or others, the Publisher does not guarantee the accuracy, adequacy, or completeness of any information and is not responsible for any errors or omissions or the results obtained from use of such information. Readers should be particularly aware of the fact that the Internet is an ever–changing entity. Some facts may have changed since this book went to press.

ISBN: 1–931841–16–0
Library of Congress Catalog Card Number: 20011098165
Printed in the United States of America
02 03 04 05 RI 10 9 8 7 6 5 4 3 2 1

Publisher: Stacy L. Hiquet
Marketing Manager: Heather Buzzingham
Managing Editor: Sandy Doell
Project Editor: Amy Pettinella
Editorial Assistant: Margaret Bauer
Technical Reviewer: David Talbot
Copy Editor: Kate Talbot
Interior Layout: William Hartman
Cover Design: Mike Tanamachi
CD–ROM Producer: David Talbot
Indexer: Johnna VanHoose Dinse
Proofreader: Lisa Neal Shaw

To Heather, Elizabeth, Matthew, and Jacob

Acknowledgments

Thanks first to Him from whom all life flows.

Heather, you work harder at these books than I do. I appreciate your sacrifices and your love more than ever. Thanks also to Jacob, Elizabeth, and Matthew for understanding why Daddy was typing all the time.

A special thank you to everyone at Premier. This group has shown its character in the time it took to produce this book. I appreciate those I know about, and the many others whose work goes unseen.

Thank you especially Stacy Hiquet for getting me started on this project, and to Amy Pettinella for her help and encouragement. Thanks to Kate Talbot for turning my mush into something readable, and for laughing at my jokes before she deleted them.

I can't thank David Talbot enough for his dual role as technical editor and CD–ROM producer. His advice and insight make this a far better book than it otherwise would have been.

A very special thanks to the Spring 2002, CSCI 490 class at IUPUI. You never complained about being guinea pigs, you worked from my very raw manuscript, and you taught me far more than I was able to teach you.

About the Author

Andy Harris began his teaching career as a high school special education teacher. During that time, he taught himself enough computing to do part–time computer consulting and database work. He began teaching computing at the university level in the late 1980s as a part–time job. Since 1995 he has been a full–time lecturer in the Computer Science Department of Indiana University/Purdue University–Indianapolis (IUPUI), where he manages the Streaming Media Lab and teaches classes in several programming languages. His primary interests are Java, Microsoft languages, Perl, JavaScript, and dynamic HTML, virtual reality, portable devices, and streaming media.

License Agreement/Notice of Limited Warranty

By opening the sealed disc container in this book, you agree to the following terms and conditions. If, upon reading the following license agreement and notice of limited warranty, you cannot agree to the terms and conditions set forth, return the unused book with unopened disc to the place where you purchased it for a refund.

License:

The enclosed software is copyrighted by the copyright holder(s) indicated on the software disc. You are licensed to copy the software onto a single computer for use by a single concurrent user and to a backup disc. You may not reproduce, make copies, or distribute copies or rent or lease the software in whole or in part, except with written permission of the copyright holder(s). You may transfer the enclosed disc only together with this license, and only if you destroy all other copies of the software and the transferee agrees to the terms of the license. You may not decompile, reverse assemble, or reverse engineer the software.

Notice of Limited Warranty:

The enclosed disc is warranted by Prima Publishing to be free of physical defects in materials and workmanship for a period of sixty (60) days from end user's purchase of the book/disc combination. During the sixty–day term of the limited warranty, Prima will provide a replacement disc upon the return of a defective disc.

Limited Liability:

The sole remedy for breach of this limited warranty shall consist entirely of replacement of the defective disC. IN NO EVENT SHALL PRIMA OR THE AUTHORS BE LIABLE FOR ANY other damages, including loss or corruption of data, changes in the functional characteristics of the hardware or operating system, deleterious interaction with other software, or any other special, incidental, or consequential DAMAGES that may arise, even if Prima and/or the author have previously been notified that the possibility of such damages exists.

Disclaimer of Warranties:

Prima and the authors specifically disclaim any and all other warranties, either express or implied, including warranties of merchantability, suitability to a particular task or purpose, or freedom from errors. Some states do not allow for EXCLUSION of implied warranties or limitation of incidental or consequential damages, so these limitations may not apply to you.

Other:

This Agreement is governed by the laws of the State of California without regard to choice of law principles. The United Convention of Contracts for the International Sale of Goods is specifically disclaimed. This Agreement constitutes the entire agreement between you and Prima Publishing regarding use of the software.

Introduction

Overview

Every so often in the programming world, a new idea is introduced that threatens to change everything. Although this is often a matter of hyperbole, the reality is that the world of programming changes with dizzying speed. If it is difficult for practicing programmers to stay current with the latest language developments, it might seem impossible for beginning programmers to work with the latest and most powerful languages.

Microsoft promises a groundbreaking development with the introduction of the .NET architecture. This programming environment clearly has the potential to be a major player in the programming universe. The .NET framework promises all kinds of things that advanced programmers have been clamoring for, such as a simplification of the C++ syntax, an easy–to–use object model, and integration of databases into programming languages. However, the languages of the .NET framework are not only for advanced programmers. Many of the innovations of C# make it an ideal starting place for beginning programmers. C# is much safer and simpler to start with than many of the other variations of C, and it has a visual interface and powerful editor that provide tons of help. The features that make C# a more advanced language often make it simpler to learn, not more complex.

I will show you some serious programming, but you're going to have a lot of fun along the way. C# is a powerful, professional language, but learning it doesn't have to be boring. I'll teach you to program the same way that I learned—by writing games.

Games are a practical, yet fun way to learn how to program, because they are motivating and interesting. Games also enable you to explore some fascinating concepts that you don't always see in other forms of programming.

Even though you will be writing a lot of games, I'll be sure to show you a lot of more serious programming concepts along the way. You'll learn how each of the concepts can be applied to real–world programming.

The best way to learn programming is to write programs. You shouldn't simply read this book; you should also use your computer. Look at the source code from the CD–ROM. Change the code around. Kick the tires a little bit. Try the challenges I give you at the end of each chapter. Use the examples to spark your interest and write something all your own.

If you do these things, I promise you that by the end of the book, you'll know a lot about the process of programming. You'll also have a firm foundation of the .NET framework and the C# language.

I had a lot of fun writing this book, and I'm looking forward to hearing from you when you succeed, so turn the page and get started!

Chapter 1: Basic Input and Output: A Mini Adventure

Programming is not something you learn by reading. You learn programming only by writing programs. In this chapter, you get started by writing a simple (silly) adventure game. You also get the basic concepts behind programming in general and C# in particular. In addition to learning how C# is organized, you learn how to

- Write the simplest interface for a C# console program.
- Write data to the screen.
- Get information from the user.
- Create basic variables.

Project: The Mini Adventure

The game at the end of this chapter is simple but fun. Showing you the game in progress is easier than describing it, so take a look at Figures 1.1 and 1.2, which show the game in progress. The computer asks the user a few questions and then makes a silly story based on the user responses.

Select C:\csab\csab01\adventure\bin\Debug\adventure.exe
Simple Adventure Game
What is your name? Jacob
What is your occupation? wanderer
Please tell me your favorite animal: aardvark
What is the name of one of your friends? Matthew
Name a problem you might face: angst
Name a tool:

Figure 1.1: The game begins by asking the user a few questions.

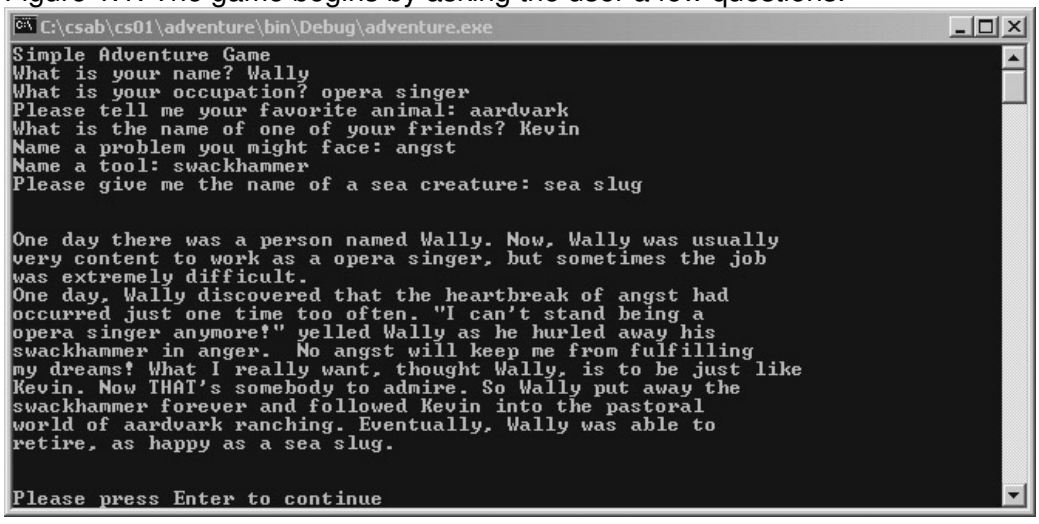

C:\csab\cs01\adventure\bin\Debug\adventure.exe
Simple Adventure Game
What is your name? Wally
What is your occupation? opera singer
Please tell me your favorite animal: aardvark
What is the name of one of your friends? Kevin
Name a problem you might face: angst
Name a tool: swackhammer
Please give me the name of a sea creature: sea slug
One day there was a person named Wally. Now, Wally was usually
very content to work as a opera singer, but sometimes the job
was extremely difficult.
One day, Wally discovered that the heartbreak of angst had
occurred just one time too often. "I can't stand being a
opera singer anymore!" yelled Wally as he hurled away his
swackhammer in anger. No angst will keep me from fulfilling
my dreams! What I really want, thought Wally, is to be just like
Kevin. Now THAT's somebody to admire. So Wally put away the
swackhammer forever and followed Kevin into the pastoral
world of aardvark ranching. Eventually, Wally was able to
retire, as happy as a sea slug.
Please press Enter to continue

Figure 1.2: The user's answers result in a silly story.

You can see that the game asks the user questions and then incorporates the answers to create a silly story. This game probably won't sell a million copies, but it's quite impressive for a first

program. After reading this chapter, you will be able to write something like it.

Note that the user interface is Spartan—no flashy graphics or eye−catching buttons and menus. For now, you are concentrating on the underlying concepts. Those other elements will come soon enough, but they add complications to your life (which you don't need just yet).

Reviewing Basic C# Concepts

The C# language was designed to profit from the experiences of other programming languages. The basic concepts behind C# programming are apparent in even the simplest programs. Essentially, a C# program can be thought of as an onion with a bunch of layers (see Figure 1.3).

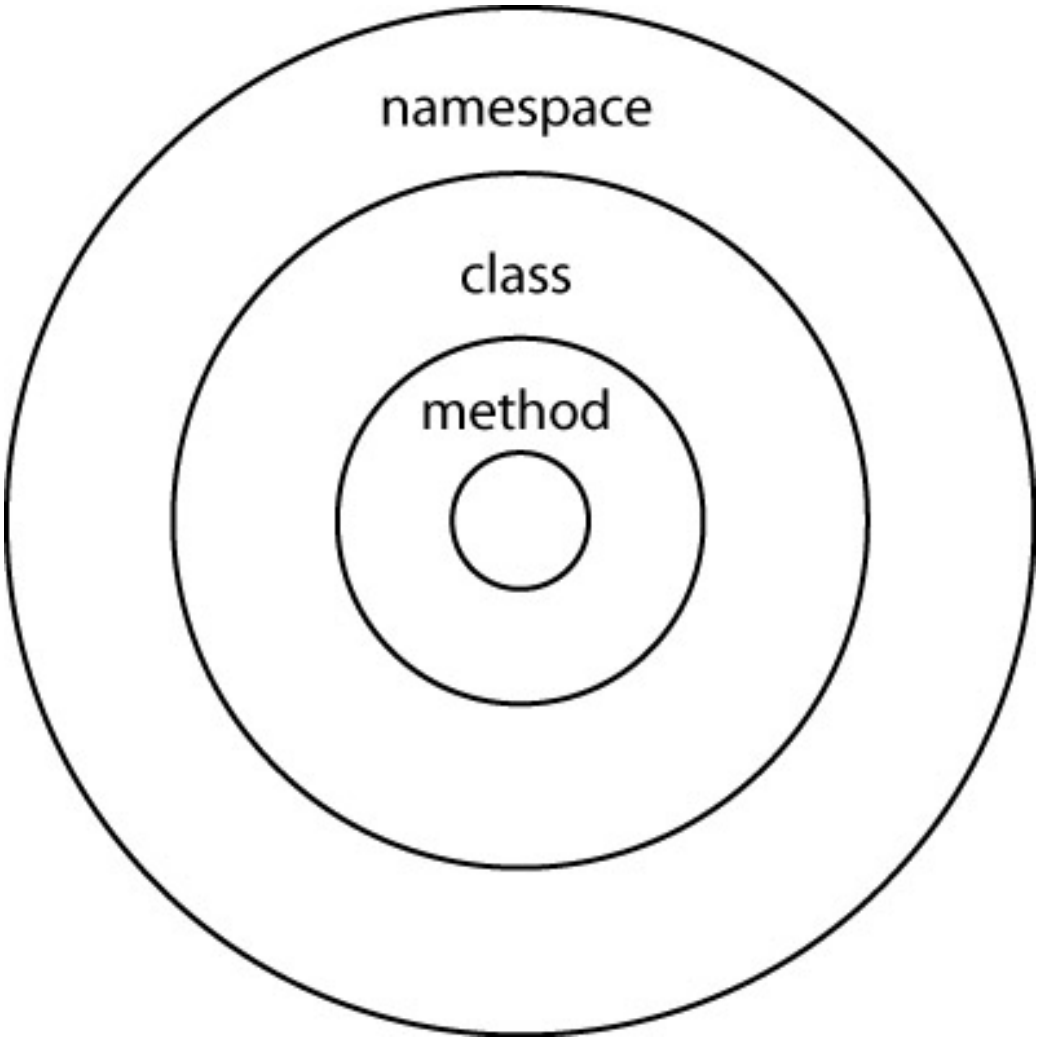


Figure 1.3: In C# programming, you have code inside methods, which are inside classes, which are inside namespaces

In the .NET environment (of which C# is a primary language) are layers of code that go from general to specific. The outer, most general, layer is the namespace. Inside a namespace, you find a series of classes, which contain methods, which contain statements.

Trap Actually, this is a simplified view. As you progress through this book (and beyond), you will see that the .NET model contains other elements. However, this reduced version of the model will suffice for now.

Namespaces

The various layers of programming help you organize your programs. Even as a beginner, you need to understand a little bit about the various layers because even the most rudimentary programs use them. Think of the layers as something like an address on an envelope. When you address an envelope, you write specific information, such as the house number. You also put the street name, which is more general, and the state, which is broad. The post office can deliver your letter by getting it to the correct state, then the correct city, then the right part of the city, and finally the specific house. Namespaces in the C# language work very much like this.

The largest landscape in the C# universe is a namespace. You can think of a namespace as a state in the postal analogy. A *namespace* is an element that enables you to group together a series of other things. Each project you create is usually a namespace. In addition, all the various things you can use in your programs—including the computer system itself, and Windows elements, such as text boxes and buttons—are separated into namespaces. Frequently, you specify which namespaces you want to work with, for example, to define whether a program should use Windows forms or a special library of math functions. If all this seems unclear to you, don't worry about it. Soon you will see examples that make it clear.

Classes

A namespace is usually made up of one or more classes. A *class* is a definition for a specific kind of object. Throughout the entire book, you will be learning about classes and objects, but essentially, they are used to describe some type of entity.

Anything a computer can describe (a database, a file, an image, a cow, whatever) can be encoded as an object. The things an object can do are called its *methods*, and the characteristics of an object are called its *properties*. Don't worry, there isn't a quiz on all this theory. You do need an introduction to these concepts, though, because all of C# is based on the idea of objects.

Methods

Classes always have methods. A *method* is a structure that contains instructions. All the commands in a program are housed in various methods of objects. Most programs have a special method named Main() (method names always end in parentheses), which is meant to execute as soon as the program begins running. If you are familiar with other languages, such as C or Visual Basic, you will see that methods are a lot like functions or subprograms in those languages.

Statements

Inside a method, you write the instructions you want the computer to execute. A *statement* is an instruction. Many statements (sometimes also called commands) involve using methods of built–in objects. Of course, a computer scientist wouldn't usually say *using* a method, because everyone would understand that. Often C# folks will refer to the process as *invoking* a method. Maybe at dinner tonight rather than asking somebody to pass the salt, you could say "Could you please invoke the salt shaker object's pass method?" It should liven up the conversation. Other commands are built in to the structure of the language.

Trick Don't worry if all this talk about methods and namespaces is making you dizzy. You don't have to memorize all this now, but you will be using it later. Even the simplest program uses all these levels of instruction, so you need to have some idea of these terms. However, you probably won't fully understand them until you build a few custom namespaces, classes,

and methods down the road. Everybody spends time in confusion until the larger picture becomes clear.

The Console Object

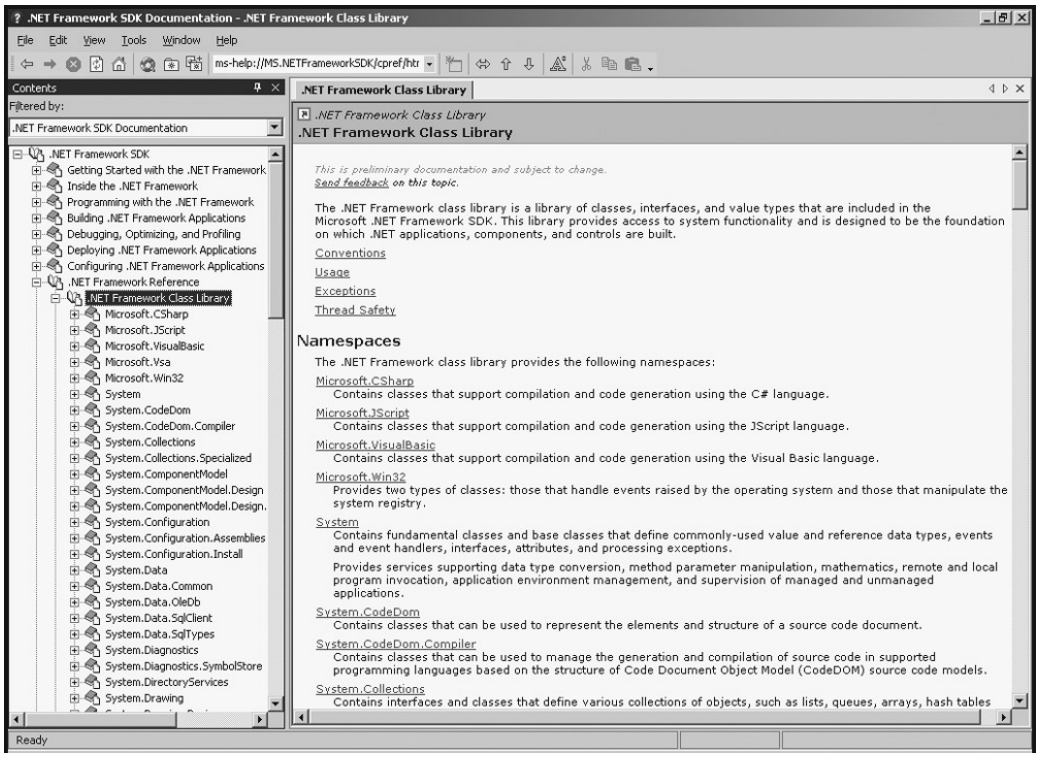
To see how all this works, take a look at one specific object, the console. In the bad old days of computing before visual interfaces like Windows, all interaction with a computer was done through a plain text screen. The combination of the text screen and the keyboards is usually referred to as the *console.* Although programming on the console might seem kind of old–fashioned, it's a good place to start because programs which feature the console are easier to write than the fancier programs using Windows forms. In C#, everything is an object, so you'll work with the console by working with a special object, also called the *Console*. Note that the names of classes are capitalized, so when I'm referring to the actual Console, class, I use a capital C. Most of your early programs will be built using the Console object, so taking a look at how C# sees this object is a good idea. If you remember working in DOS or command–line UNIX, you probably have some fond memories of the console. Most console applications use only text and appear only in black and white. Modern programs for end users don't usually work with the console because it makes things much more difficult for users who prefer menus, buttons, and toolbars. However, knowing how to program on the console is still useful because some applications don't require a graphic user interface, such as server–side programs in Web development, code libraries, and simple applications. The main reason I'm starting you out on the console, though, is that it's a much easier place to program. Although all those graphic elements make the user's life easier, they can cause headaches for beginning programmers.

> **Trick** In the earlier days of computing, all computing happened on a simple black–and–white text screen. It was an easy way to learn programming because you had fewer things to learn (and fewer things could go wrong). Programming on the console is still a very important skill, and because it's still a relatively easy place to work, you start there in your programming journey. You will be able to write programs that look more familiar to a Windows user or a Web surfer as you progress through this book, but all the main ideas can be demonstrated using the generally simpler console.

The console itself can be thought of as a DOS window. If you've been around computing for a while, you probably remember the days when you had to type all your commands into a text–only window. The Console object is the way C# views that window, which is still available in modern computing, and is surprisingly useful. To do anything useful with the console, you need to know how to use the Console class within C#, which ships with documentation describing all the various parts of the language. Looking through this documentation will also give you a sense of how the language is organized.

.NET Documentation

To understand the general layout of the language, take a look at Microsoft's official documentation for the .NET framework. This should be installed on your machine as part of the Visual Studio environment, but it may appear as a separate element in the Start menu. (On my machine, it is Programs, Microsoft .NET framework SDK, Documentation.) Figure 1.4 shows this screen in action.


.NET Framework SDK Documentation - .NET Framework Class Library
File Edit View Tools Window Help
ms-help://MS.NETFrameworkSDK/cpref/htr
Contents
Filtered by:
.NET Framework SDK Documentation
.NET Framework SDK
Getting Started with the .NET Framework
Inside the .NET Framework
Programming with the .NET Framework
Building .NET Framework Applications
Debugging, Optimizing, and Profiling
Deploying .NET Framework Applications
Configuring .NET Framework Applications
.NET Framework Reference
.NET Framework Class Library
Microsoft.CSharp
Microsoft.JScript
Microsoft.VisualBasic
Microsoft.Vsa
Microsoft.Win32
System
System.CodeDom
System.CodeDom.Compiler
System.Collections
System.Collections.Specialized
System.ComponentModel
System.ComponentModel.Design
System.ComponentModel.Design.
System.Configuration
System.Configuration.Assemblies
System.Configuration.Install
System.Data
System.Data.Common
System.Data.OleDb
System.Data.SqlClient
System.Data.SqlTypes
System.Diagnostics
System.Diagnostics.SymbolStore
System.DirectoryServices
System.Drawing
.NET Framework Class Library
.NET Framework Class Library
.NET Framework Class Library
This is preliminary documentation and subject to change.
Send feedback on this topic.
The .NET Framework class library is a library of classes, interfaces, and value types that are included in the Microsoft .NET Framework SDK. This library provides access to system functionality and is designed to be the foundation on which .NET applications, components, and controls are built.
Conventions
Usage
Exceptions
Thread Safety
Namespaces
The .NET Framework class library provides the following namespaces:
Microsoft.CSharp
Contains classes that support compilation and code generation using the C# language.
Microsoft.JScript
Contains classes that support compilation and code generation using the JScript language.
Microsoft.VisualBasic
Contains classes that support compilation and code generation using the Visual Basic language.
Microsoft.Win32
Provides two types of classes: those that handle events raised by the operating system and those that manipulate the system registry.
System
Contains fundamental classes and base classes that define commonly-used value and reference data types, events and event handlers, interfaces, attributes, and processing exceptions.
Provides services supporting data type conversion, method parameter manipulation, mathematics, remote and local program invocation, application environment management, and supervision of managed and unmanaged applications.
System.CodeDom
Contains classes that can be used to represent the elements and structure of a source code document.
System.CodeDom.Compiler
Contains classes that can be used to manage the generation and compilation of source code in supported programming languages based on the structure of Code Document Object Model (CodeDOM) source code models.
System.Collections
Contains interfaces and classes that define various collections of objects, such as lists, queues, arrays, hash tables
Ready
Figure 1.4: Here's the .NET documentation. I've expanded the tree on the left to show the various namespaces available in the .NET environment.

Trick If the .NET documentation is not available on your machine, you should install it before going much further. It is a road map to all of C#, and your way will be much easier if you have access to this map.

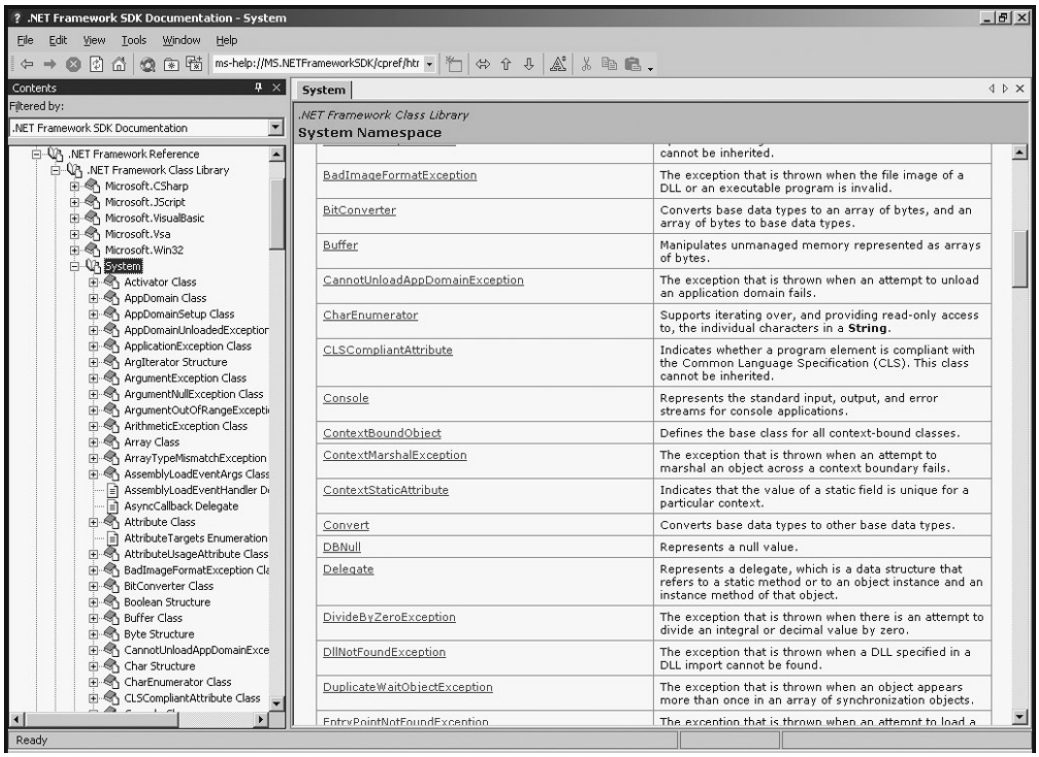
A huge amount of information is in the .NET documentation, but you don't need to concern yourself with all of it. For now, I just want you to see what's there. The right panel shows a long (intimidating) list of namespaces available to you as a programmer. When the documentation first comes up, you won't see much in the right–hand panel, so, click the System link under Namespaces to see the contents of the System namespace.

In the Real World

You might be confused about the relationship between C# and .NET. This confusion is understandable because the two technologies are very closely intertwined. *.NET* is Microsoft's term for its new programming architecture. The basic idea of .NET is to have several languages use the same underlying architecture, which should have a natural relationship with the various forms of the Windows operating system. Most of Microsoft's next generation of programming languages, including the latest editions of C++ and Visual Basic, use the .NET environment. However, C# is the first major language designed from the beginning with .NET in mind. Because of this, many pundits speculate that C# will be the most commonly used language in the .NET universe. All programmers in the Microsoft world (there *are* other kinds of programming) will probably have to learn some form of the .NET model, so C# is a natural choice because of its close relationship with the model. Throughout this book, when you learn about specific syntax issues (such as where to put semicolons and how the assignment operator works), you're actually learning the C# language. When you learn about certain objects, such as the **Console** object or command buttons, you're learning about the .NET universe. If you don't see the distinction yet, that's okay. Just note that if you ever want to learn another .NET language (such as Visual Basic, or *VB*), you will find it an easy jump because both C# and VB use the.NET framework. The .NET framework also provides some interesting possibilities for Internet programming, but these techniques do not work on every web server.

The System Namespace

As you can see in Figure 1.5, the System namespace consists of many (again, intimidating) classes. Each of these classes represents an object you can use to write your programs. For now, you can safely ignore most of them, but there is a class to represent the console. Click the appropriate link to examine the Console class. The page of text you see is almost useless, but at the bottom of that page is a link named *Console Members.* Click this link to learn about the characteristics of the Console class and the things it can do.


.NET Framework SDK Documentation - System
File Edit View Tools Window Help
Contents
Filtered by:
.NET Framework SDK Documentation
.NET Framework Reference
.NET Framework Class Library
Microsoft.CSharp
Microsoft.JScript
Microsoft.VisualBasic
Microsoft.Vsa
Microsoft.Win32
System
Activator Class
AppDomain Class
AppDomainSetup Class
ApplicationException Class
ArgIterator Structure
ArgumentException Class
ArgumentNullException Class
ArithmeticException Class
Array Class
ArrayTypeMismatchException
AsyncCallback Delegate
Attribute Class
AttributeTargets Enumeration
AttributeUsageAttribute Class
BitConverter Class
Boolean Structure
Buffer Class
Byte Structure
Char Structure
CharEnumerator Class
CLSCompliantAttribute Class
System
.NET Framework Class Library
System Namespace
cannot be inherited.
BadImageFormatException
The exception that is thrown when the file image of a DLL or an executable program is invalid.
BitConverter
Converts base data types to an array of bytes, and an array of bytes to base data types.
Buffer
Manipulates unmanaged memory represented as arrays of bytes.
CannotUnloadAppDomainException
The exception that is thrown when an attempt to unload an application domain fails.
CharEnumerator
Supports iterating over, and providing read-only access to, the individual characters in a String.
CLSCompliantAttribute
Indicates whether a program element is compliant with the Common Language Specification (CLS). This class cannot be inherited.
Console
Represents the standard input, output, and error streams for console applications.
ContextBoundObject
Defines the base class for all context-bound classes.
ContextMarshalException
The exception that is thrown when an attempt to marshal an object across a context boundary fails.
ContextStaticAttribute
Indicates that the value of a static field is unique for a particular context.
Convert
Converts base data types to other base data types.
DBNull
Represents a null value.
Delegate
Represents a delegate, which is a data structure that refers to a static method or to an object instance and an instance method of that object.
DivideByZeroException
The exception that is thrown when there is an attempt to divide an integral or decimal value by zero.
DllNotFoundException
The exception that is thrown when a DLL specified in a DLL import cannot be found.
DuplicateWaitObjectException
The exception that is thrown when an object appears more than once in an array of synchronization objects.
Ready
Figure 1.5: Some classes in the System namespace. The Console has features for communicating with the user that will be helpful.

The Console Class

The Console is a simple (but important) class. Like most classes, it has properties (which you will ignore for now) and methods (shown in Figure 1.6). *Methods* are the tasks that the Console object knows how to do. You want to do one thing in this program—write a message to the user. Fortunately, the Console class contains several methods designed to do exactly that. Take a careful look at Write().

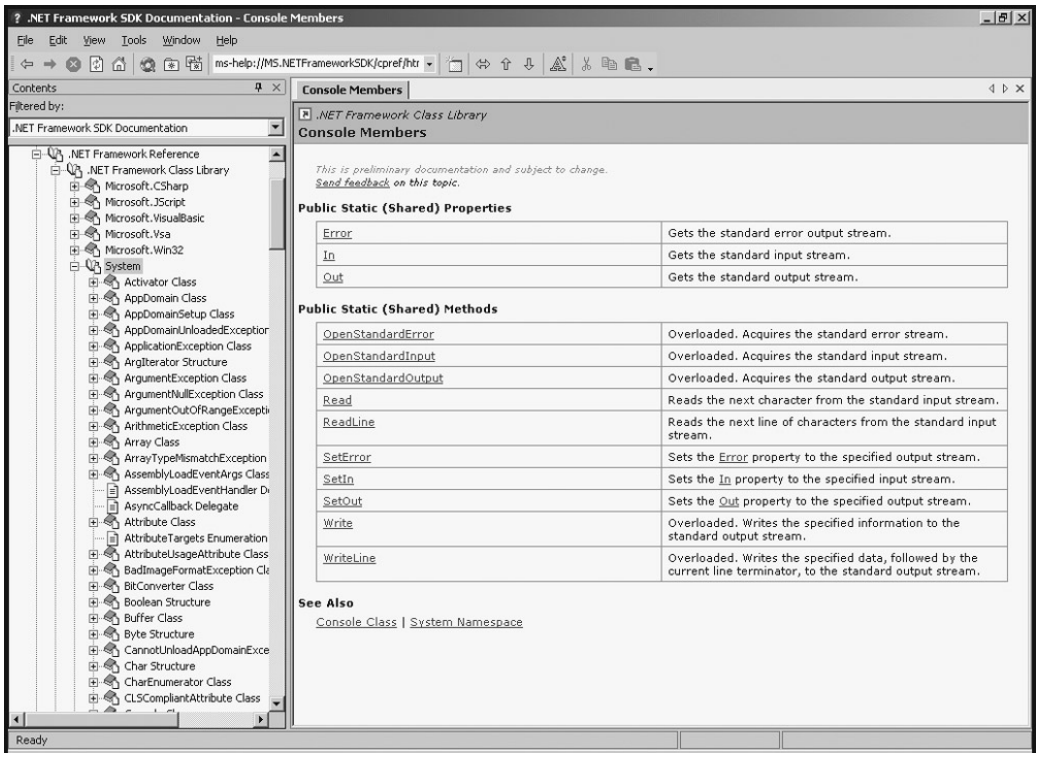


Figure 1.6: The members of the Console class.

The Write() Method

The Write() method enables you to write a message to the text screen. Anything you want to write to the screen should be enclosed in quotes inside the parentheses. If you want to write *Hi, Mom!*, use this command:

```
System.Console.Write("Hi, Mom!");
```

This command demonstrates the entire hierarchy of structures in C#. System is a namespace, which contains the Console class, which contains the Write() method. This is cumbersome enough to warrant a loophole. If you use the command

```
using System;
```

at the beginning of your program, you no longer have to specify System, and you can simply write

```
Console.Write("Hi, Mom!");
```

The WriteLine() Method

The Write() method has an even smarter cousin, named WriteLine(). The easiest way to explain the difference between them is with a demonstration.

This code fragment

```
Console.Write("Hi, ");
Console.Write("Mom!");
```

appears on the console as Hi, Mom!.

Trap Console.Write does not add anything to the text. Note that I include a space after the comma in "Hi, ". Without the space, the output would be Hi,Mom!.

Each invocation of Console.Write() causes the new text to be written at the next spot on the screen, usually on the same line. Often, you are generating one line of text at a time. The Console.WriteLine() method is used to write text as a complete line, adding a new line (like pressing the Enter key in a word processor) to the end of the line. Here's an example:

```
Console.WriteLine("Hi");
Console.WriteLine("Mom");
```

Output:

```
Hi
Mom
```

Although knowing about the Write() and WriteLine() methods is helpful, understanding how to get around in the documentation is even more important. Whenever you want to accomplish a task in C#, usually you can find a method attached to an object in a particular namespace that will fulfill your needs.

Saying "Hello, World!"

The programming world has a surprising number of well–established traditions. One of them is the Hello World program, which is the first program you write in any new environment. It simply pops up on the screen and says, Hello, World!. This is a fun tradition but also has a practical side. It is usually the simplest kind of activity you can make a computer do in a given language. By starting with such a simple program, you can focus your efforts on becoming comfortable with the programming environment. With a debugging and programming package as complex as Visual Studio, starting with a simple program so that you can get your feet wet in the environment makes a lot of sense.

The Hello World program featured in Figure 1.7 doesn't do much, but it illustrates several important ideas in programming. When you understand the code behind this very simple program, you will have a framework that can be reused for every C# program you write.

Figure 1.7: As advertised, the program says "Hello, World!"

Getting into the Visual Studio .Net Environment

Although writing C# programs using any text editor is possible, you will probably spend most of your time using the Visual Studio Integrated Debugging Environment (IDE). The Visual Studio IDE is based on earlier Microsoft languages, notably Visual Basic and Visual C++. One interesting feature of the .NET version of the IDE is that the same environment is used to program many languages. This is consistent with the tighter integration that now exists between the Microsoft languages. Now there are fewer differences between programs written in different languages in the Microsoft universe.

After Visual Studio .NET (sometimes referred to as *Visual Studio 7*) is loaded onto your machine, you activate it as you would any other program—from the Start menu.

As you can see from Figure 1.8, the IDE is a very complicated beast. Don't worry about understanding the whole thing at once. I'll show you the various parts as you need them. For now, rely on your experience as a software user. It's reasonable to guess that the icons represent the most commonly needed functions in the program and that all the major commands are available through the online menu system. You might want to hover your mouse over the screen icons to find the important ones (such as the New Project button). For the most part, you write programs in the large gray area in the center of the screen. Everything else on the screen gives you information about what's going on in the program or gives you access to tools such as the command line and various windows components. Because you aren't going to use those features yet, you can leave them alone for now.

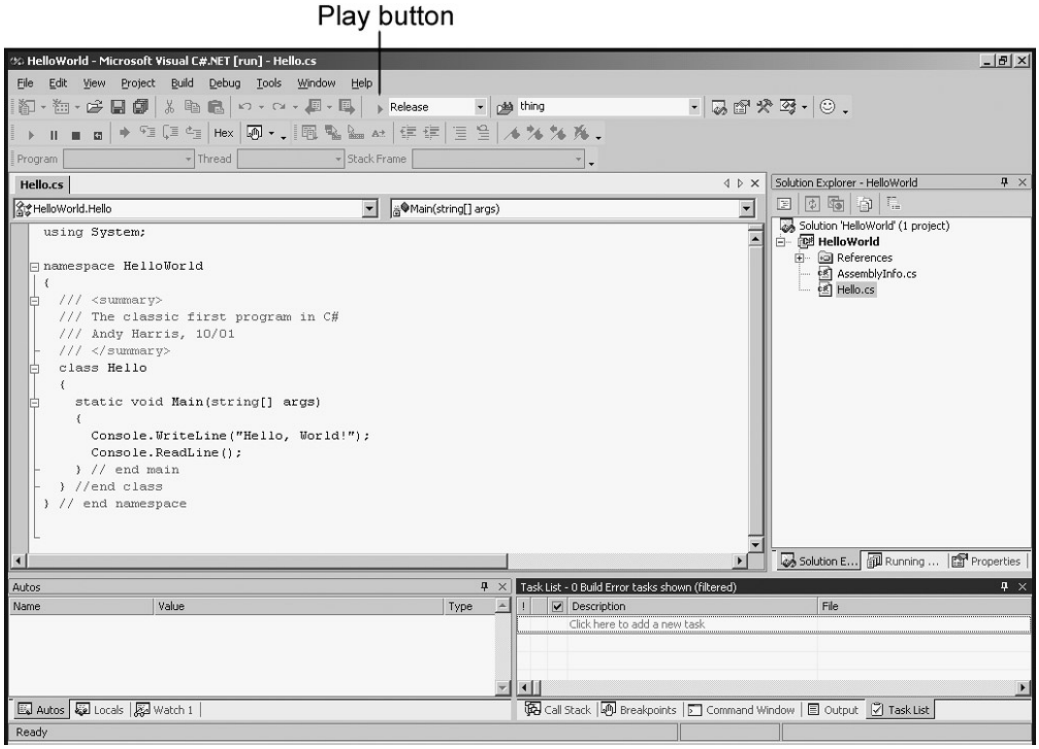


Figure 1.8: The Visual Studio IDE as it appears on my computer.

Starting a New Project

Start a new project by clicking—you guessed it—the New Project button, which lives in the upper–left corner of the screen. If you are averse to buttons, you can choose New, Project from the File menu. In either case, you see a dialog box that looks like Figure 1.9.

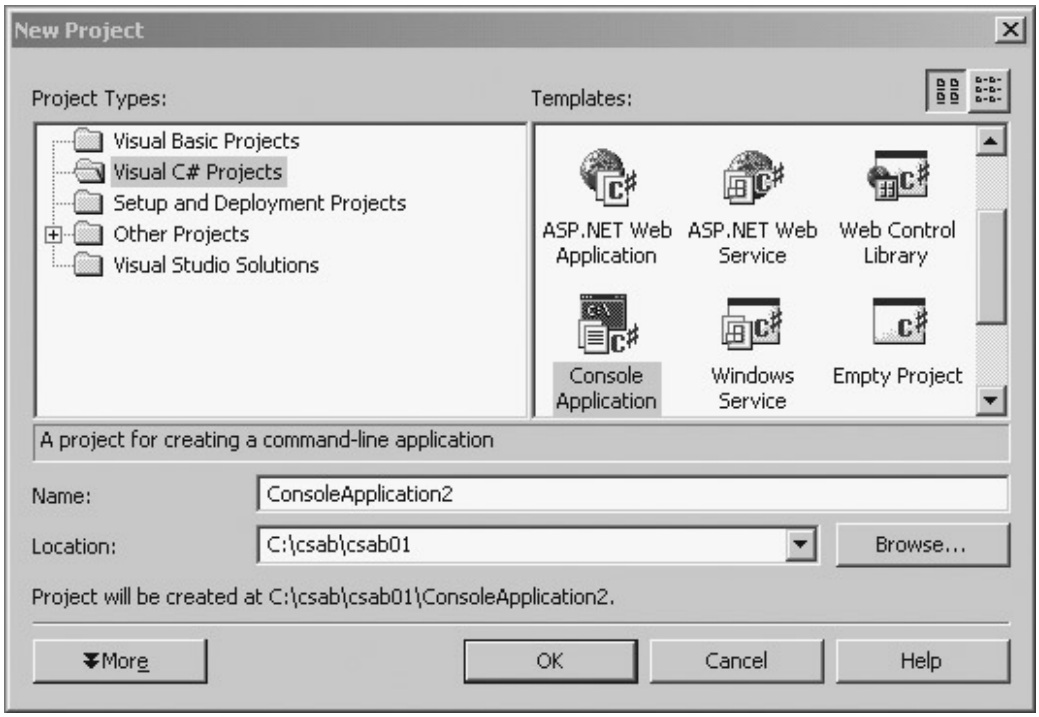


Figure 1.9: The New Project dialog box is where you determine the programming language, the project's, and the type of project your are writing.

The New Project dialog box in Figure 1.9 has many important features. For example, the Project Types list box on the left enables you to determine which programming language you want to use. Depending on the way Visual Studio is configured on your system, you might have several other options. I currently have my machine configured for Visual Basic and C#. (I use other languages, too, but not usually in the .NET framework. Somehow it seems rude to use a Microsoft environment to write Perl code.) For the programs in this book, you always choose the C# environment.

Choosing the Project Type

After selecting the programming language, you can choose the type of project. You can use C# to write many types of programs. In the early stages of this book, you will write console applications, which are a simple interface because they are the easiest to understand. After you learn the basics of C# with these simple interfaces, you will graduate to Windows applications and eventually Web applications. For now, choose Console Application. However, be sure that you name your application and choose a location for it before pressing Enter or double–clicking the Console Application icon.

> **Trap** If you double–click the Console Application icon before choosing a name or location for your project, Visual Studio assigns you a default name and location. It can be a real pain to fix this after the fact, so be sure that you type in a name and location before pressing Enter or clicking OK. I've made this mistake a number of times.

Examining the Code Window

After you determine the general characteristics of the program, the IDE starts writing code for you. All programs of a certain type share certain characteristics, so Visual Studio will supply boilerplate code to get you started. You can think of the automatically generated code as an outline that you can flesh out to write your program.

Figure 1.10 displays the code window as it appears after a new project named *HelloWorld* is

created. All the critical parts of any C# program are present, and the program will run, although it doesn't do anything interesting yet.

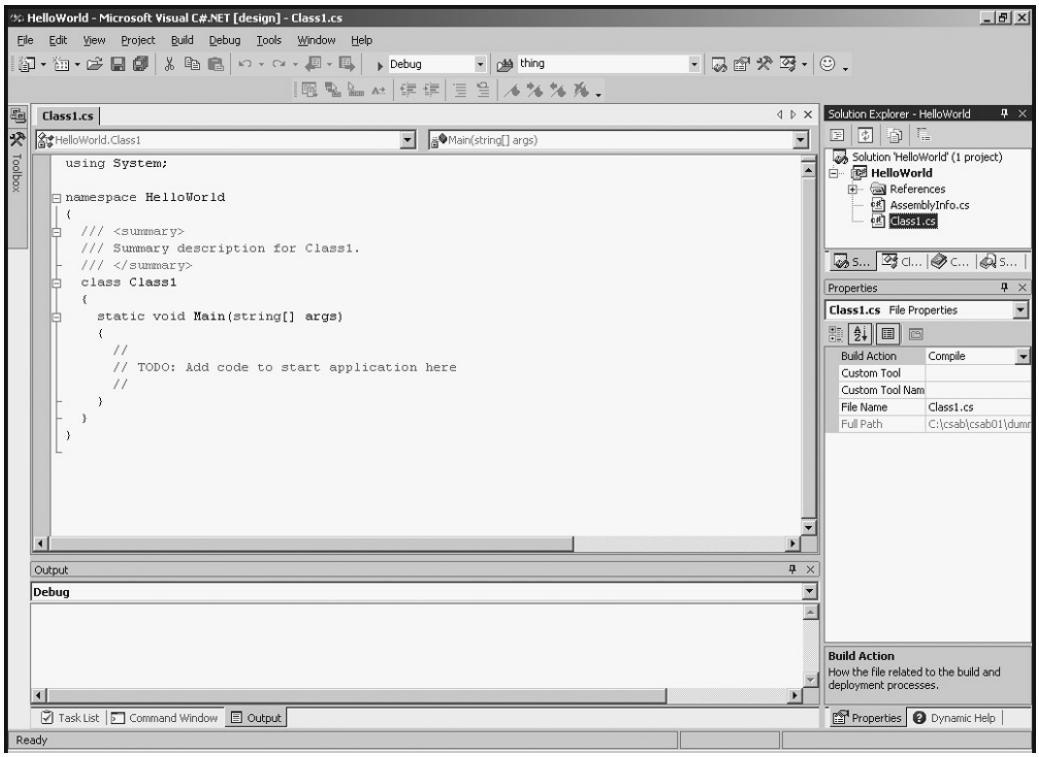


Figure 1.10: The HelloWorld program displayed in the code window.

You have to learn a few things about C# before you start studying the code. Although Figure 1.10 doesn't show it, the code is displayed in several colors. Words appear in blue, black, green, and gray. The colors indicate the type of information the compiler thinks each word is. For example, comments are in gray.

Also, you will note a certain symmetry to the text. Towards the beginning of the code are several left braces ({). Later in the code, you see matching right braces (}). The braces are used to group lines together. (I promise to show you exactly how. For now, I just want you to see the gestalt of the language so that you will understand later how the details fit together.) The braces are carefully matched so that every left brace has a right brace aligned directly underneath it (although sometimes several lines below the left brace) and everything inside the braces is indented. This is a common way of writing code in the languages derived from C, and because the IDE automates this style of code, you will stick with it now.

Trick A passionate discussion about vertically aligning your braces is ongoing in programming circles. To tell the truth, most languages (including C#) completely ignore the spacing and indentation in your code. The spaces help the programmer, not the computer. I prefer a different indentation convention, but because this form is built–in to the editor and is a reasonably standard approach, I will go with it for this book. The most important thing is to have a consistent style and stick with it. As you will see, indentation, commenting, and the like, can have a major effect on how well you get your programs to work.

You will also see minus signs to the left of the editor. When you click one of these symbols, you "collapse" the braces that follow the indicated line. This helps you to look at specific parts of your program and hide unnecessary details.

Examining the Default Code

As I just mentioned, the IDE starts to build your code for you. For your part, you will begin by examining the boilerplate code and later will add a little functionality. Here's code that Visual Studio created:

```
using System;

namespace HelloWorld
{
	/// <summary>
	/// Summary description for Class1.
	/// </summary>
	class Class1
	{
		static void Main(string[] args)
		{
			//
			// TODO: Add code to start application here
			//
		}
	}
}
```

This code is the same for any console–style application you write. Visual Studio gives you a starting place so that you don't have to begin with a completely blank page. If you choose a different kind of application (like the Windows applications or Web applications you will write later in this book) the IDE will generate.

Adding a Reference to a Namespace

The first line given by the IDE says using System. The using statement indicates that a program will be using commands from a specific namespace. In a sense, the idea of namespaces is already familiar to you. At home, my wife calls me Andy. Calling me Andy Harris would be silly because everybody in our house is named Harris. At my job, there's another guy named Andy, so people are more likely to say Andy Harris when they want to talk to me. You can always use a first name and a last name, but at home, your last name is implied.

Referring to a Namespace with a Using Statement

The using statement in C# works in a similar way. It enables you to use a group of commands that are related. You will see many namespaces in future chapters, but almost every program written in C# uses the System namespace because it contains useful objects. You need the console later, and the console object's full name is System.Console. If you use the using System statement at the beginning of your program, you can simply refer to Console instead of System.Console. Almost every program in C# starts with the using System statement. As you learn more about C#, you will learn about other namespaces you will want to include in your programs.

Creating a Custom Namespace

The namespace HelloWorld line is used to generate your own namespace. In addition to the namespaces built in to the .NET environment, each project you create can have its own namespace. By default, the editor builds a namespace based on the project's name. The namespace is called HelloWorld but actually contains all the code on the screen. You can see the left brace immediately after the namespace line. All the code is then indented until the

corresponding right brace. This indentation scheme helps you remember that all the interior code is part of the namespace.

Adding Summary Comments

Right after the namespace definition, you see three lines that begin with three slashes (///). Lines that begin this way are used to create documentation for your programs. Generally, you leave alone the lines containing <summary> and </summary> and, between these lines, add text that describes your project. This description of your program is stored along with your program. One advantage of C# is that programs are supposed to have some of the documentation built in. Any comments you put between the summary tags will be part of this automatic documentation. Of course, if you don't add comments, the automatic documentation feature cannot work.

Creating the Class

Class1 defines a class. Essentially, a class is a way of grouping your code. For now you can think of a program as a class because your early programs will have one namespace and one class. As you get more sophisticated, you'll build namespaces with multiple classes. Classes are the key to C# programming. Right now, the HelloWorld namespace has one class in it, Class1. Actually, the official name of the class is HelloWorld.Class1, but because you are inside the HelloWorld namespace, you don't have to worry about specifying the namespace. Generally, one of the first things you do when creating a program is rename your class. As a programmer, you get many opportunities to name things. Give your class a name that describes what the program does. Later in this chapter, you will change the class name from Class1 to Hello. Class names in C# usually start with a capital letter.

Like the namespace, a class definition begins a new part of the code and has a pair of braces to denote the new structure.

Trick Whenever you create a new program, be sure to change the name of the class. Although the program will run without changing the name, you will find this confusing later, especially when your programs have a number of classes.

Examining the Main() Method

static void Main(string[] args) begins the Main() method. Any code inside this pair of braces automatically executes when the program is run. For now, all the code in your programs will go inside the Main() method.

Trap Watch your capitalization, especially if you're accustomed to other C languages. C# uses a capital *M* in *Main*, but most other variants of C use a lowercase *m*.

I will explain later what all the parts of the Main command are, so don't be intimidated by the string[] args). For now, you don't need to worry about these details because the editor will build this line for you. You can concentrate, instead, on customizing this code to make it do something interesting.

Examining the Rest of the Code

Inside the Main() method, you see three lines that begin with two slashes (//). Any line that begins with these slashes is a *comment*. The compiler ignores comments. However, comments are among the most important aspects of good programming. You use comments to *document* your code—to explain something that's going on or to make a note to yourself. The comments here tell you where

you will write the actual code. You will delete these comment lines and replace them with program code.

You also see a series of right braces. Each of these right braces is vertically aligned with its corresponding left brace. If you don't include all the right (closing) braces, your program will not work correctly.

Modifying the Code

Although the IDE creates all this code for you, the first part of writing a C# program is to make changes to the code you're given. You have to make a number of changes right away. Take a look at my modified version of the code:

```
using System;

namespace HelloWorld
{
      /// <summary>
      /// The classic first program in C#
      /// Andy Harris, 10/01
      /// </summary>
      class Hello
      {
            static void Main(string[] args)
            {
                  //
                  // TODO: Add code to start application here
                  //
            } // end main
      } //end class
} // end namespace
```

I made a number of small but important changes to the program. First, I added comments to the summary section. At a minimum, you should add comments (to remind yourself what the program is supposed to do), your name, and the date. This might seem like a silly exercise, but it's a very good habit to form. Note that these summary comments begin with three slashes.

Next, I changed the name of the class from Class1 to Hello. Hello is a much better name for the class because it is more descriptive than Class1.

For the time being, I left the content of the Main() method (the comments with the TODO note in them) alone. I'll change those soon, but first, there's some more housekeeping to do.

You might want to add comments after every right (closing) brace because you will have many of these braces in your C# travels, and it's easy to get confused. Because you use the same character to end a namespace, method, and class definition, figuring out exactly what you intended to end can be a challenge. Not every programmer does this, but I think that it's a terrific habit to cultivate, especially when you're getting started.

Writing to the Console

At this point, your program still doesn't do what it's supposed to do—greet the user. Now you are ready to change the code in the Main() method. Take out those comment lines and add the following line of code:

```
Console.WriteLine("Hello, World!");
```

This line of code sends the message Hello World to the console, which is another way of saying *to the DOS prompt*. (Remember, you are beginning with DOS–based programs because they are simpler, but you will graduate to Windows–style programs soon enough.)

On the next line, write the following code:

```
Console.ReadLine();
```

This line causes the program to stop and wait until the user presses Enter. If you don't add this line (or something like it), the program will stop running and disappear before the user can read the greeting.

Trick As soon as you type the period after Console, a list of possible completions appears in the editor window. You can use the arrow keys to look at the entire list and the Tab key to choose the selected element. Because the Console is part of the System namespace, the editor knows all the terms that can be associated with it and gives you an easy way to choose legal terms to finish the statement. When you write the left parenthesis, you see a similar little window explaining which kind of data should go in the parentheses. These little helper windows prevent mistakes by giving you hints on the syntax of C#.

Placing Semicolons Correctly

As you look at the code, you see a semicolon (;) at the end of some lines but not others. You can spot one at the end of the using line and the Console lines but not in the other lines in the program. A pair of braces follows most of the other lines in the program. These indicate that the line begins a structure, such as a namespace, class, or function. The set of braces and whatever they contain are considered a part of that line. Many of the other commands (such as the two I mentioned at the beginning of this paragraph) do not begin a structure. To tell the compiler that you are finished writing a particular command, you must end the line with a semicolon (like writing a period at the end of a sentence to indicate that the sentence is finished).

Most of the time, a semicolon appears at the end of a line of C# code. The only time this is *not* true is when

- The next character is a left brace.
- The current character is a right brace.
- You are writing one logical line of code on more than one line on the text editor.

Don't get hung up on memorizing these rules. After you write a few programs, placing semicolons will be a snap.

Moving from Code to a Program

If you don't get to see it working, your program isn't really a program, so it's time to compile your program. Writing programs usually happens in a series of steps: You design the program, write it, test it, and refine it. So far, you've designed the program (the design of this program couldn't get much simpler), and you've written it.

Your complete program looks like this:

```
using System;
```

```
namespace HelloWorld
{
      /// <summary>
      /// The classic first program in C#
      /// Andy Harris, 10/01
      /// </summary>
      class Hello
      {
            static void Main(string[] args)
            {
              Console.WriteLine("Hello, World!");
              Console.ReadLine();
            } // end main
      } //end class
} // end namespace
```

Now you must test your program.

Compiling Your Program

Ultimately, all that computers can manipulate are binary on and off values—everything the computer does boils down to these elements. Everything the computer knows how to do is expressed in a small list of commands called *opcodes* that are built in to the hardware of the machine. Even these instructions are expressed in binary form. You can write a program by entering those numbers directly into the computer in binary notation. (In fact, that's exactly how the first home computer, the Altair, was operated.)

However, this kind of programming, called *machine language programming*, is tedious and error–prone. The computer can work well with a program in machine language, but writing machine language properly is very difficult for programmers. Computer scientists devised programming languages to make the job easier. Although the syntax of a language such as C# is not much like English, C# is far easier for a programmer to understand and use than machine language. However, computers cannot work directly with the code written in C# or any other high–level language. For the computer to do anything with your program, your program has to be translated (*compiled*) into machine language.

Fortunately, the Visual Studio IDE makes compiling and running your programs simple. Click the blue arrow that looks like a VCR play button, near the top center of the IDE screen. If all goes well, you will see a black screen that looks like Figure 1.11.

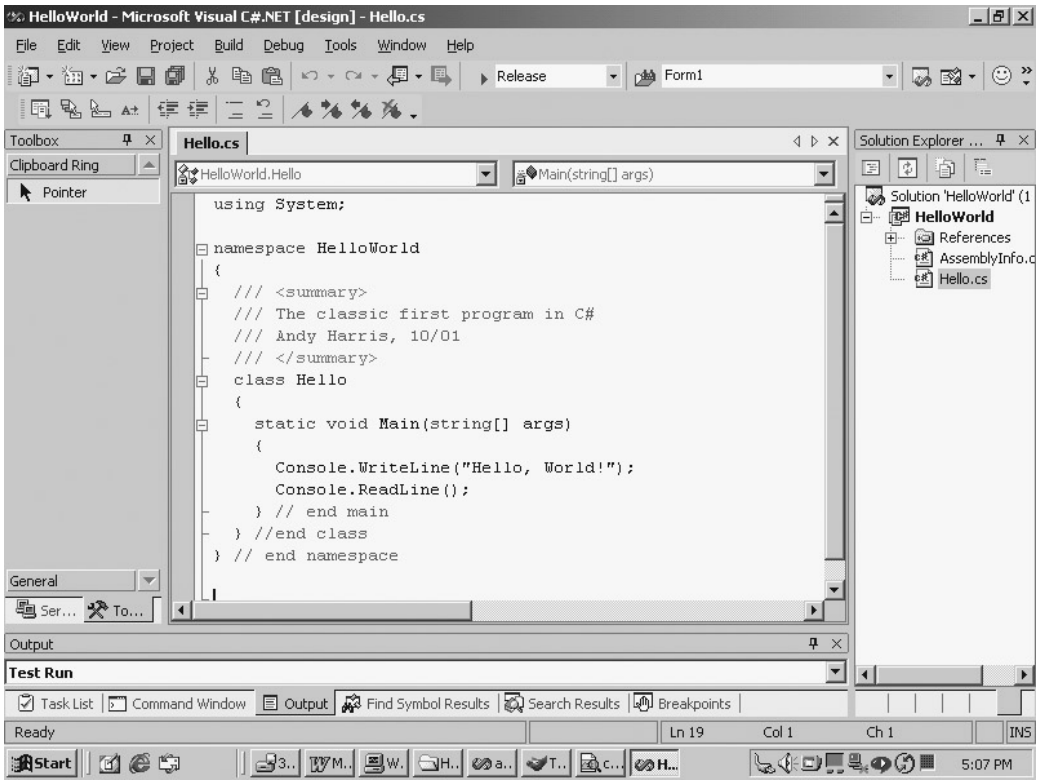


Figure 1.11: The Play button compiles and runs your program.
The black screen featured in Figure 1.12 is the console.

Figure 1.12: A cheerful greeting from the Hello World program.
Congratulations! You have written your first program. If you look around on the menu structure created by the C# environment, you will see that you have a HelloWorld.exe file. If you double–click that file, your program will run. (Cool, huh?)

Looking for Bugs

Programs usually do not work on the first try. Many things can go wrong. Simple typing mistakes and errors in logic cause all kinds of problems, even to experienced programmers. Fortunately, if things go wrong, C# has many tools to help you make things right. For example, if I forget to put the semicolon after the Console.WriteLine(); statement in the Hello World program, the editor places a red squiggle at the end of the line (much like the spell checker in Word). When I try to compile the program, I get an error message in the build menu (see Figure 1.13).

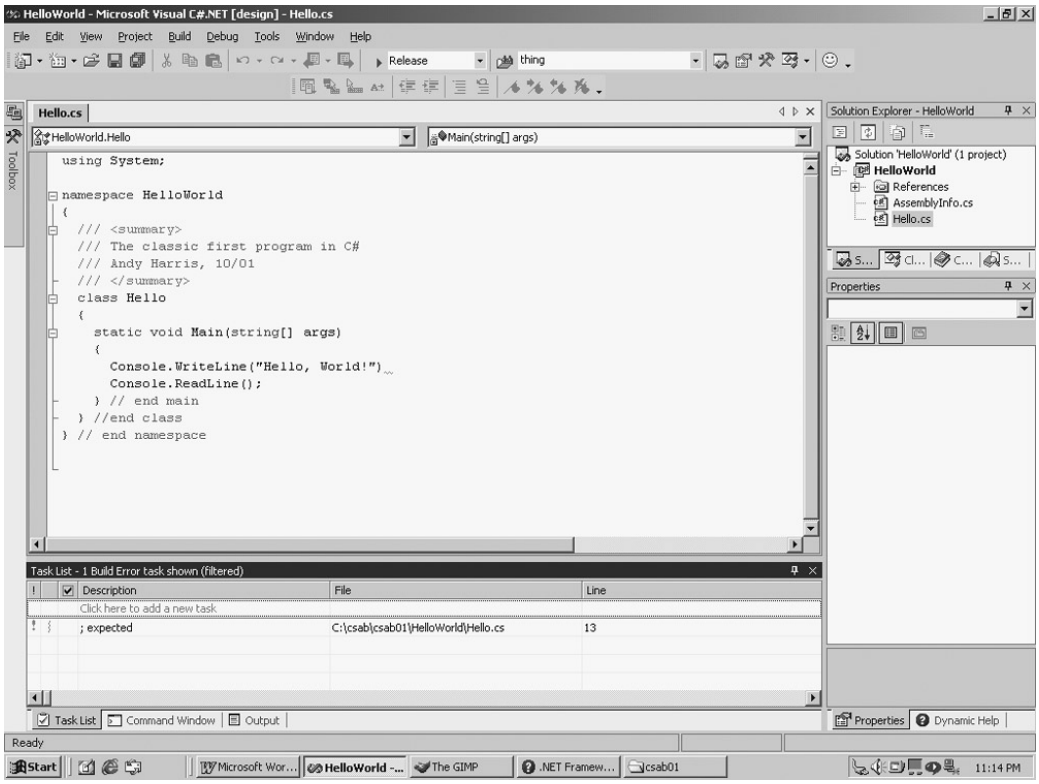


Figure 1.13: The squiggle at the end of the WriteLine() command indicates a missing semicolon.

When you try to run a program that contains errors, C# informs you that there are build errors and asks whether you want to run anyway. Generally, you say no so that you can fix those errors. Any errors that C# notices are placed in a task list at the bottom of the screen. By clicking an item in this list, you are automatically taken to the appropriate line in the code.

Trap The compiler reports where it noticed the error, which isn't always where the error is located. Still, it gives you a decent hint about what went wrong.

If a program does not compile correctly, don't panic. Look at the task list and try to solve each problem in order. Often, solving one problem automatically solves the others.

Getting Input from the User

Being able to write information to the screen is very nice, but computer programs are supposed to be interactive. It is even better if the program can get input from the user. Take a look at the program featured in Figure 1.14 to see an example of a simple program that interacts with the user.

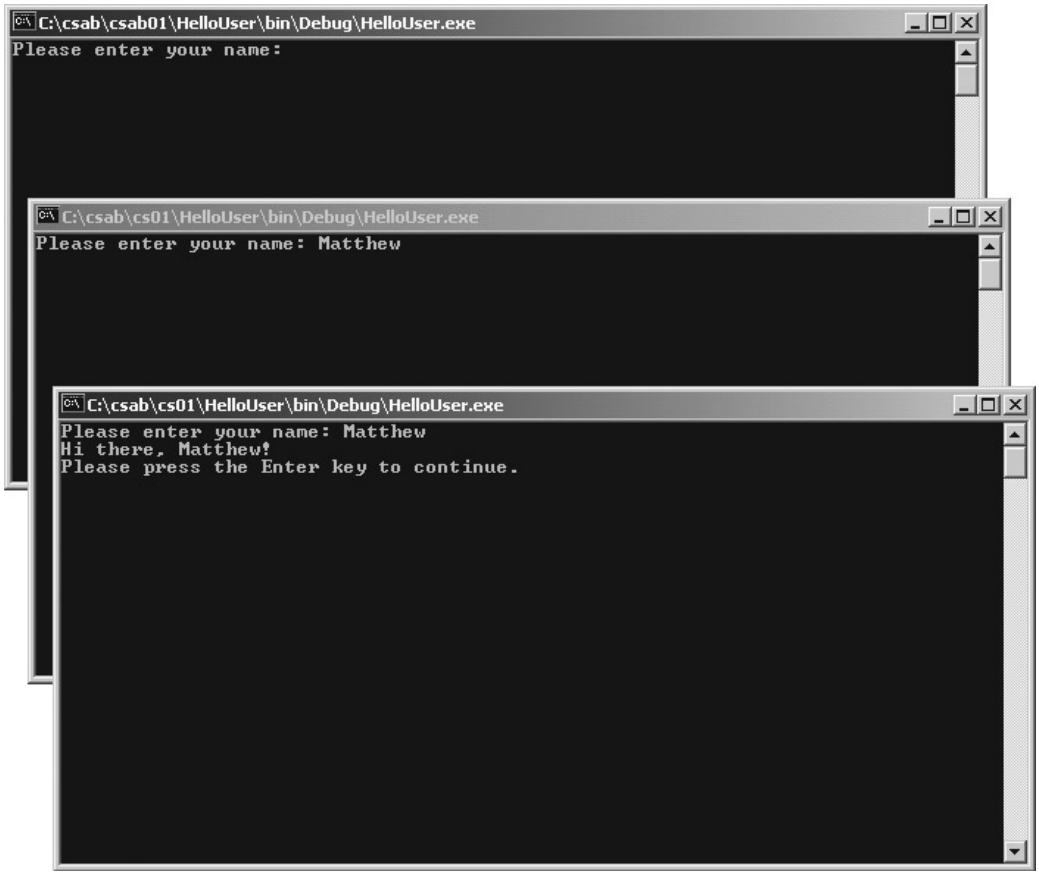


Figure 1.14: The user types a response and receives a customized greeting.

Getting a value from the user is a straightforward task but requires you to understand a couple new concepts. First, look at the code, and you will see that it is very similar to the Hello World program:

```
using System;

namespace HelloUser
{
  /// <summary>
  /// Add user input to the Hello World program
  /// Andy Harris, 10/30/01
  /// </summary>
  class Hello
  {
    static void Main(string[] args)
    {
      string userName;
      Console.Write("Please enter your name: ");
      userName = Console.ReadLine();
      Console.WriteLine("Hi there, {0}!", userName);
      Console.WriteLine("Please press the Enter key to continue");
      Console.ReadLine();
    } // end main
  } // end class
} // end namespace
```

Even though the code is mainly familiar, a couple elements might have caught your eye. These changes are all used to add interactivity. This version of the program will ask the user for a name and will use that name in a personalized greeting. To do this, you need a new concept called a *variable* and you'll use the Console.ReadLine() method in a new way.

Creating a String Variable

If you're going to ask the user for something, you must have a place ready to catch it. Ultimately, computers store all information in memory cells, which handle only binary information. Even seasoned programmers generally prefer to work directly with numbers and text instead of the binary values the computer understands. Computer languages allow you to create special places in memory, designed to hold information. These memory cells are *variables.* You will deal with many kinds of variables as a C# programmer, but one of the most interesting types is text. Of course, computer scientists could never call this kind of information *words* or *text* because everybody would know what they are talking about. Instead, text information is almost always called *string data* in computing circles.

Trick Actually, text is referred to as *strings* for a descriptive, almost poetic reason. Computers can't deal with words at all, or even letters. A letter is stored as a numeric value, using a code such as ASCII (or, in later languages such as C#, unicode). Text is simply a bunch of these numeric values placed in contiguous cells, like beads on a string. All this isn't important, I suppose, but it is cool to wander around muttering about string manipulation under your breath. People might think that you're smart.

The line

```
string userName;
```

is simply setting up a chunk of memory so that it is ready to store text data. The term string is used to tell the compiler to set up a memory area to handle string (or text) values. The term userName refers to the name I have given this piece of memory.

Trick As a programmer, you will have many opportunities to name things. A few guidelines might come in handy:

- Don't use spaces or punctuation in names; these can potentially confuse the compiler.
- Use descriptive names. If you don't, you could find yourself baffled about its meaning when you come back to debug it. Naming the variable something more descriptive, such as *radius* or *taxRate*, is far better.
- Resist using long variable names because misspelling them is too easy, which could cause problems later.
- C# is a case–sensitive language. Most programmers use mainly lowercase letters in their variable names, reserving uppercase letters for differentiating words (such as the capital *R* in *taxRate*).

Getting a Value with the Console.ReadLine() Method

Now that you have a variable (the string variable userName) to hold a string value, you need to get that value from the user. The ReadLine() method of the console object is used to, well, read a line from the console. It waits for the user to type something on the screen and, as soon as it encounters the Enter key, returns whatever was typed. Notice the way the ReadLine() method is written in the program:

```
userName = Console.ReadLine();
```

This line of code is an example of an *assignment statement*. Assignment is one of the most important ideas in programming, but it's very simple. Some sort of value is being copied from one thing to another. As you read this line, train yourself to think of the equal sign (=) as *gets*, not *equals*. This is important because in C#, the equal sign is used *not* to determine equality but as an assignment operator. (That statement will make more sense after you read Chapter 2, "Branching and Operators: The Math Game.") If you read the line as "userName gets Console.ReadLine()," you will understand what this line of code is supposed to do. It tells the computer to get a line of text from the console and copy that text value to the string variable userName. In most programming languages, assignment flows from right to left. That is, the variable (userName) is given the value (whatever is read from the console).

Incorporating a Variable in Output

After the ReadLine() code is placed in memory named userName, containing whatever text the user typed. The next step is to print out this value to the user as a customized greeting.

The line that provides the greeting looks like this:

```
Console.WriteLine("Hi there, {0}!", userName);
```

If you compare this line to the output, you can probably figure out what's going on. The computer says, "Hi there," places the user's name in place of the {0} stuff, and adds an exclamation point to the end. The WriteLine() method can be used to combine plain text with variables. It works by first expecting a line of text. If you want to add variables in your message, you can replace a variable with a number inside braces. Computers usually start counting at zero, so userName is variable number zero, and the value of userName is printed out to the screen. If you ask for a first name and a last name, the line might look like this:

```
Console.WriteLine("Hi there, {0} {1}!", firstName, lastName);
```

If you also incorporate a middle initial, the code might end up like this:

```
Console.WriteLine("Hi there, {0} {1}. {2}!", firstName, mi, lastName);
```

As you can see, the plain text you want to write should be added first, with placeholders for any variables you might want to include in the message. Then you provide a list of variables. Of course, the order of the variables in the list can make a big difference, and if you refer to variable number 1, you must have at least two variables in the list.

> **Trap** Computers begin counting at zero! The first element in a list is *not* number one but number zero. Forgetting this is easy if you're new to programming! Most of the time in your writeLine() statements, you will simply be referring to variable zero ({0}), or you will have no variables at all. By the way, if you want to know the fancy computer scientist name for placing the variables inside the text, it's *string interpolation*. See whether you can work that phrase into your dinner conversation tonight.

Combining String Values

The ability to write string values to the screen is very useful, but you should know about a couple other special circumstances. Sometimes you want to combine string variables in other ways. Also, you often want to type special characters, such as the tab character or quotation marks, to the screen or force a carriage return at a specific place. Take a look at the program in Figure 1.15, which illustrates some interesting printing problems.

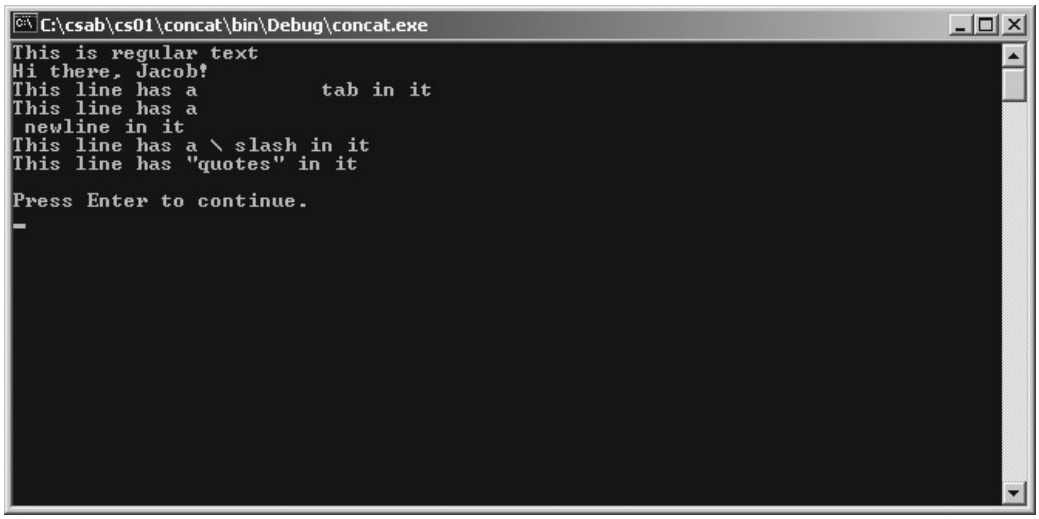

C:\csab\cs01\concat\bin\Debug\concat.exe
This is regular text
Hi there, Jacob!
This line has a tab in it
This line has a
newline in it
This line has a \ slash in it
This line has "quotes" in it
Press Enter to continue.

Figure 1.15: This program demonstrates several interesting problems.

Take a look at the source code for this program. You will see that it demonstrates techniques that can be very useful as you write characters to the screen:

```
using System;

namespace concat
{
  /// <summary>
  /// Demonstrates string concatenation,
  /// escaped characters
  /// </summary>

  class concat
  {
    static void Main(string[] args)
    {
      string userName = "Jacob";
      Console.WriteLine("This is regular text");
      Console.WriteLine("Hi there, " + userName + "!");
      Console.WriteLine("This line has a \t tab in it");
      Console.WriteLine("This line has a \n newline in it");
      Console.WriteLine("This line has a \\ slash in it");
      Console.WriteLine("This line has \"quotes\" in it");
      Console.WriteLine();
      Console.WriteLine("Press Enter to continue");
      Console.ReadLine();
    } // end main
  } // end class
} // end namespace
```

The program consists of several lines written to the screen. The first line is typical, but each of the others illustrates a different technique for printing to the screen.

Combining Strings with Concatenation

The program has a string variable named userName. You can see that I have printed out the value of the variable, but I used a technique different from the one described earlier in the chapter. You can use a plus (+) sign to combine literal string variables (whatever is contained inside quotes) and string variables.

Computer scientists like to create complicated names for simple concepts, and this is no exception. Using plus signs like this to combine string objects is called *string concatenation*.

Trick Having two ways to do the same thing might seem strange, but it makes a lot of sense. For ordinary situations, you often use the interpolation trick shown at the beginning of this chapter, but in certain situations, concatenation makes more sense.

Adding a Tab Character

Sometimes you will want to send information to the screen that would be easy with a keyboard, but not so simple when you are writing a program. For example, the next line of code looks like this:

```
Console.WriteLine("This line has a \t tab in it");
```

You can see the \t combination, which is a backslash followed by a *t* character. This special combination stands for *tab*. Whenever the compiler encounters this sequence, it acts as if the Tab key was pressed. If you look up at the output of this program, you see a gap where the \t combination was placed in the original code.

Using the Newline Sequence

C# allows you to use some other special characters. Perhaps the most useful is the *newline* character, which is the combination \n. Whenever the compiler sees this sequence, it adds a carriage return. If you look again at the output, you see that the line breaks exactly where the \n sequence occurs in the original string.

Hint Console–based applications (such as the ones in this chapter) do not have any sort of word wrap. If you are writing a long complicated string to the screen, you might have to insert newline sequences to ensure that the lines are separated appropriately.

Displaying a Backslash

Because the backslash is used in all these special sequences, you might wonder how you display the backslash character. All that's necessary is to include two backslashes together, as I did in this line:

```
Console.WriteLine("This line has a \\ slash in it");
```

Displaying Quotation Marks

Sometimes you want to show quotation marks in text. This can be difficult because the quote symbol is also used to determine where the string begins and ends.

You might guess the solution—simply precede the quote symbol with a backslash to indicate that you want to display a quotation mark.

```
Console.WriteLine("This line has \"quotes\" in it");
```

Launching the Mini Adventure

You now know enough to write the adventure story mentioned at the beginning of this chapter. The program itself is reasonably simple. However, there is a process to building the program.

Planning the Story

To write this program, I started by writing the silly story on paper and circling the words I thought would be fun to replace with the user's responses. I then created a variable for each of those words.

Trick Because the code for this program is a little longer than the earlier programs, I have divided it into parts so that I can describe each part of the program individually. You can see the entire program on the CD−ROM that accompanies this book. In fact, I encourage you to load this project (and all the others in this book) from the CD so that you can see them in the editor and modify them for your own use.

Creating the Variables

The first part of the game involves all the standard procedures: creating a namespace, a class, the Main() method, and variables. Here's the code that performs these tasks:

```
using System;

namespace adventure
{
  /// <summary>
  /// Silly Adventure game
  /// User responds to some questions, and these
  /// responses are used to write a goofy story
  /// Andy Harris, 11/02/01
  /// </summary>
  class Adventure
  {
    static void Main(string[] args)
    {
      string person;
      string occupation;
      string seaCreature;
      string animal;
      string friend;
      string tool;
      string problem;
```

Notice that I added some comments at the beginning to help myself remember what the program is supposed to do. I named the namespace and the class and created all the variables I thought I would need.

Hint Unlike some languages, C# does not require you to declare all your variables at the beginning of the Main() method, but it's still a good practice. Describing all the

variables in one place where you can see them together is very handy.

Getting Values from the User

After creating the variables, you get values from the user for those variables. Each variable is loaded up in much the same way, by asking the user a question with a Console.Write() method and getting a value with the Console.ReadLine() method. Here's the code that asks all the questions and stores the responses in variables:

```
  Console.WriteLine("Simple Adventure Game");
  Console.Write("What is your name? ");
  person = Console.ReadLine();

Console.Write("What is your occupation? ");
occupation = Console.ReadLine();

Console.Write("Please tell me your favorite animal: ");
animal = Console.ReadLine();

Console.Write("What is the name of one of your friends?);
friend = Console.ReadLine();

Console.Write("Name a problem you might face: ");
problem = Console.ReadLine();

Console.Write("Name a tool: ");
tool = Console.ReadLine();

Console.Write("Please give me the name of a sea creature: ");
seaCreature = Console.ReadLine();
```

You might be surprised that I chose to use Console.Write() instead of Console.WriteLine() to ask the questions. I tried them both and preferred the behavior of Console.Write() in this case. If I had used WriteLine(), there would have been a carriage return at the end of the line, and the user's response would have been typed on the next line. (If you don't know what I'm talking about, change one of the Write statements to a WriteLine, and you will see the effect.) I think that it looks more like a dialog if the response is on the same line as the question, so I decided to use Write() instead of WriteLine().

For each of the variables in the story, I used a Console.ReadLine() call to get the current line of response from the user, and I stored that response in the appropriate string variable.

Writing the Output

The last element is to write the story to the screen. It probably won't surprise you to learn that I used several calls to the Console.WriteLine() method to achieve this effect:

```
//create some blank lines
      Console.WriteLine();
      Console.WriteLine();

      //Write the story
      Console.WriteLine("One day there was a person named {0}. Now, {0} was Ä
      usually ", person);
      Console.WriteLine("very content to work as a {0}, but sometimes the Ä
      job ", occupation);
      Console.WriteLine("was extremely difficult.");
```

```
Console.WriteLine("One day, {0} discovered that the heartbreak of {1} Ä
had ", person, problem);
Console.WriteLine("occurred just one time too often. \"I can't stand Ä
being a ");
Console.WriteLine("{0} anymore!\" yelled {1}, as he hurled away his ", Ä
occupation, person);
Console.WriteLine("{0} in anger. No {1} will keep me from fulfilling", Ä
tool, problem);
Console.WriteLine("my dreams! What I really want, said {0}, is to be Ä
just like", person);
Console.WriteLine("{0}. Now THAT's somebody to admire. So {1} put Ä
away the ", friend, person);
Console.WriteLine("{0} forever, and followed {1} into the pastoral" , Ä
tool, friend);
Console.WriteLine("world of {0}-ranching. Eventually, {1} was able Ä
to ", animal, person);
Console.WriteLine("retire, as happy as a {0}", seaCreature);
```

To get the story game started, I first typed the story on the screen as Console.WriteLine() statements. For example, my first draft at the first line was this:

```
Console.WriteLine("One day, there was a person named {person}. Now, {person} was");
```

Of course, this version helped me see how to set up the code, but it wouldn't compile correctly. To make that happen, I had to modify the code so that the variables to interpolate follow the main string, like this:

```
Console.WriteLine("One day there was a person named {0}. Now, {0} was Ä


Usually ", person);
```

Note in this particular circumstance that I used the variable person twice, so there was no need to repeat it.

Take a careful look at this line:

```
Console.WriteLine("occurred just one time too often. \"I can't stand being a ");
```

Note that I used the backslash and quote combination (\") to get quotation marks in my story. I wanted to print a quotation mark on the screen, but if I used a regular quote symbol, the compiler could become confused because it might think that this is the end of the string. The \" sequence informs the compiler that you want to send a quotation mark to the screen, rather than use it as a programming construct.

Finishing the Program

All that remains is some cleanup. To keep the program on the screen, I'll ask the user to press the Enter key as usual. Notice the use of WriteLine() statements without any parameters. These are used to send blank lines to the console. They can dramatically improve the clarity of your output.

Here's the remaining code in the adventure program:

```
//create some blank lines
Console.WriteLine();
Console.WriteLine();
```

```
      //ask for Enter to quit
      Console.Write("Please press Enter to continue");
      Console.ReadLine();

    } // end main
  } // end class
} // end namespace
```

As usual, please notice that I added comments to all the right (closing) braces to make it easier to spot the elements I'm ending.

Summary

In this first chapter, you have come a very long way. You have become familiar enough with the basic structure of C# to start poking around in the system documentation. You have learned how to work with the IDE to generate a default console application. You can now interact with the user through a few methods of the console object, and you know how to send special sequences, such as tab and newline, to the screen. You know what a string variable is and how to make one. In short, you're now a programmer. In the next chapter, you will learn how to have the computer change its behavior with branching statements.

Challenges

- Write a program that prints your favorite quote to the screen.
- Write a program that asks for a person's first name, last name, and middle initial and then writes the name in several formats (first, middle, last), (last, first, middle).
- Write your own variant of the adventure game. Start by writing a story, and then choose key words to be substituted.

Chapter 2: Branching and Operators: The Math Game

Now that you know how to set up a basic C# program, you are ready to manipulate information in a more interesting way. In this chapter, you will learn several ways computers deal with information. You will also learn how to have computers appear to make decisions and how to generate random numbers, which are useful in a surprising number of situations. After reading this chapter, you will be able to:

- Understand numeric data types.
- Convert variables from one type to another.
- Use if and switch statements to control branching behavior.
- Create a random number.
- Understand basic math and assignment operators.

The Math Game

As usual, you will learn all these elements in the context of a simple game. For this chapter, you will write a game that is great for kids in elementary school. Figures 2.1 and 2.2 show you the interface of the Math Game.

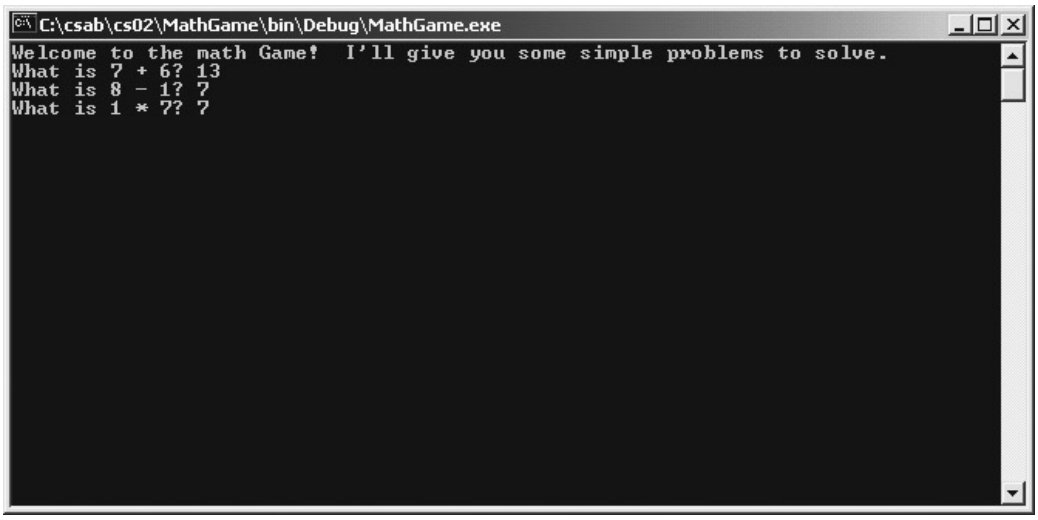

C:\csab\cs02\MathGame\bin\Debug\MathGame.exe
Welcome to the math Game! I'll give you some simple problems to solve.
What is 7 + 6? 13
What is 8 - 1? 7
What is 1 * 7? 7

Figure 2.1: The program asks the user simple math questions.

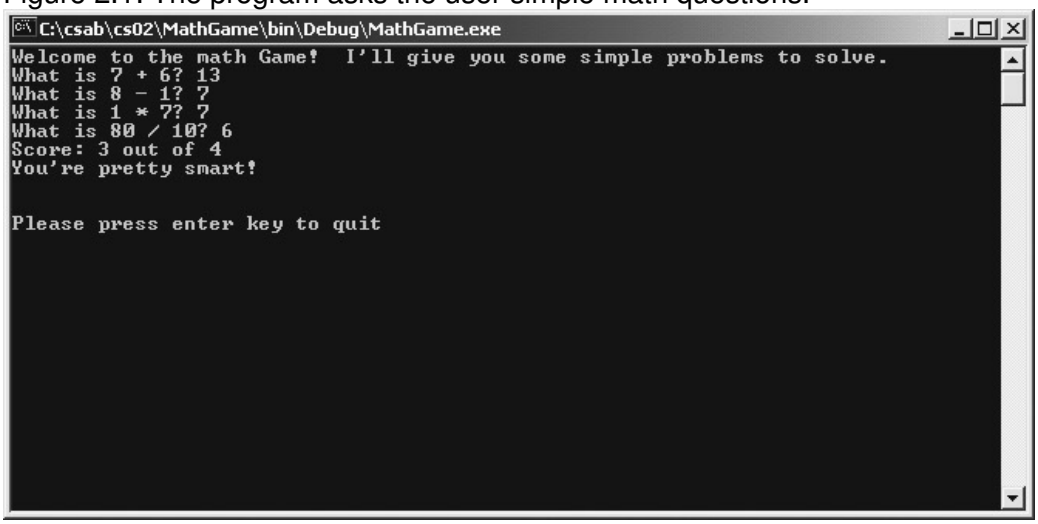

C:\csab\cs02\MathGame\bin\Debug\MathGame.exe
Welcome to the math Game! I'll give you some simple problems to solve.
What is 7 + 6? 13
What is 8 - 1? 7
What is 1 * 7? 7
What is 80 / 10? 6
Score: 3 out of 4
You're pretty smart!
Please press enter key to quit

Figure 2.2: When the user is finished, the program reports a score.

The program has features that might not be immediately apparent. To keep the program interesting, the problems are randomly generated each time the program is run. Also, because the game is for younger children, I decided that the answers should always be positive integers. (The problems are fixed so that subtractions will never come out negative, and division problems will never have a remainder.) The program keeps track of the number of questions the user answers and gives a score based on the user's responses.

Using Numeric Variables

A computer programmer needs to understand *how* computers work with information. The hardware of modern computers works with little on/off switches that work in Base–two mathematics. Some people refer to this type of mathematics as *binary mathematics* and use values of 1 and 0 instead of on and off switches. All information in a computer, from text to video games, is converted internally into binary numbers before the computer can do something useful with it. In the earliest days of computer programming, you had to work in binary notation to do anything.

Fortunately, modern languages such as C# spare you this tedium. You can tell the computer that you want to work with text (which programmers call *strings*, as you recall from Chapter 1, "Basic Input and Output: A Mini Adventure"), integers (positive and negative numbers without decimal values), real numbers (numbers with decimal values), and complex types of information, such as dates, pictures, sounds, and whatever else you can store in a computer. Although you don't need to be fluent in binary to be a computer programmer, you do need to understand that the computer uses different tricks to translate all these kinds of data into binary information.

The Simple Math Game

Computers are good at math, but you must look out for a few things. The various types of data storage can have some surprising side effects. The following program, shown in Figure 2.3, illustrates some of the things that can go wrong.

Figure 2.3: The computer can do math, but it gave some strange results here. Is 5 divided by 4 really 1?

This program is silly but it illustrates some important points about how numbers work in computing. Take a look at the source code for this program, and I'll explain what's going on:

```
using System;
```

```
namespace SimpleMath
{
  /// <summary>
  /// Demonstrates basic variable stuff
  /// Andy Harris, 11/09/01
  /// </summary>
  class DoMath
  {
    static void Main(string[] args)
    {
      int a = 1;
      int b = 2;
      float c = 2.4f;
      float d = 4.7f;
      Console.WriteLine("Math Demo");
      Console.WriteLine();
      Console.WriteLine();

      //addition with integers works as expected
      Console.WriteLine("5 + 4 = {0}", 5 + 4);
      Console.WriteLine("{0} + {1} = {2}", a, b, a + b);

      //division by integers can cause problems
      Console.WriteLine("5/4 = {0}", 5 / 4);

      //dividing by floating point values works better
      Console.WriteLine("{0} / {1} = {2}", c, d, c/d);
      Console.WriteLine("5f/4f = {0}", 5f / 4f);

      Console.WriteLine();
      Console.WriteLine();
      Console.Write("Please press \"enter\" to continue");
      Console.ReadLine();

    } // end main
  } // end class
} // end namespace
```

The design of this program is straightforward. The computer simply performs basic mathematical operations and reports the results. The program demonstrates important details about how numeric variables are created and used. I'll explain how this program works in the next few sections. First, take a look at variable types.

Numeric Variable Types

C# supports a number of numeric variable types. Each type specifies a different way the numeric data is translated into binary. Table 2.1 describes several key data types in C#.

C# supports other types of variables besides the ones listed here, but for a beginning programmer, these variable types work just fine. Each type of variable takes up a specific amount of memory and is best suited for working with a particular type of number. In general, you work with two primary types of numbers in C# programming: the int and double types.

Hint The reference to unicode in the table above refers to a scheme for storing characters in binary form in computer memory. Computers have used a scheme called ASCII for many years, but C# and a number of other languages now use the unicode standard, because it provides support for most written languages, including those which do not use Roman characters.

Table 2.1: Selected Data Types

Type	Description	Values	Examples
bool	Boolean (true/false)	True or false	true
char	One character	Characters in English or other languages (with unicode)	‘a’
int	Integer (positive or negative value without a decimal point)	–2 billion to 2 billion (roughly)	34 –7
long	Long integer	+/– $9 * 10^{18r}$	3L –23L
float	Floating–point real number	+/– $3.5 * 10^{38}$, 7 significant digits	1.5f –3.1415927f
double	Double–precision real number	+/– $1.7 * 10^{308}$, 15 significant digits	–2.5 3.1415

Integer Variables

You might remember from grade school that *integers* are positive and negative numbers and zero but not numbers that end in a decimal (or fractional) value. For example, 3, –100, and 0 are integers, but 3.14159 and –0.5 are not. Most of the time in your programming career, you will work with integer values. Although in math integers can be infinitely large or small, they aren’t in computing. Storing a number in a computer’s memory takes a finite amount of space, so the variable types are organized according to how much memory they use. The more memory a variable type uses, the larger range of numbers it can handle.

C# has two main types of integers. The int variable type uses 32 digits of binary to represent a number, meaning that an int variable values roughly between 2 billion and negative 2 billion. This range is related to the largest and smallest numbers that can be stored in 32 digits of binary math. Most of the math you do on a computer probably falls within that range, so if you don’t need a decimal point, you can make an int variable. Any time you refer to a number without a decimal point in your code, the number is interpreted as an int value. The values 5 and 4 are called *literal* values, because they are not variables. However, the computer still needs to assign a type to such values, and because there is no decimal point in the values, they will be interpreted as literal ints.

```
Console.WriteLine("5 + 4 = {0}", 5 + 4);
```

5 and 4 are int literals, but you can also create a variable that is defined as an int. Take a look at these lines in the Simple Math code:

```
int a = 1;
int b = 2;
```

The keyword int is used to indicate that the compiler should generate a variable of type int. The computer reserves the appropriate amount of memory, and then the user can refer to that chunk of memory as ‘a.’ The value 1 is immediately stored into that memory location.

For example, this code

```
Console.WriteLine("variable a is {0}", a);
```

produces the following output.

Output:

```
variable a is 1
```

Trick Although it isn't necessary, you can assign a value to a variable as you are creating it, as I did in this program. This ensures that the variable always has a legitimate value. The C# compiler complains if you create a variable that doesn't have a value assigned. If you prefer, you can simply create a variable without assigning a value, like this: int a; Then assign the value later: a = 1;

Long Integers

At times you need a value larger or smaller than the range of the int. If you need a very large or small integer, you can use the long variable type. A long integer stores its information in 64 digits of binary, so it can handle large (or, of course, small) values. If you need to create a long integer, just use the keyword long, like this:

```
long a;
```

If you want to use a long integer as a literal value (for example, assigning a value to a long variable), you use the modifier L at the end of the number, like this:

```
long a = 21L;
```

Trap Remember to use a capital *L* to specify that the value is a long integer. The lowercase *l* looks a lot like the numeral 1, which can be extremely confusing.

Floating–Point Variables

Like integers, real numbers have multiple kinds of number schemes. *Real* numbers are those numbers that include decimal values. Computer systems often store real numbers in a *floating–point notation*, meaning that the decimal point can go anywhere in a number, which gives you a lot of flexibility. Although integers are described by their *range* (their minimum and maximum values), the most important characteristic of a real number is its *precision*, the number of significant digits it supports. Floating–point numbers store their values in a form of scientific notation, so they can be extremely large or extremely small. C# uses two main types of real numbers. The float type uses 32 digits of binary and supports a large range. The double type (for the double–precision floating type) uses 64 digits and handles larger values and values extremely close to zero. You probably won't be surprised to find that you create a floating–point value by using the float keyword, like this:

```
float myFloat;
```

You refer to a float value with an f character at the end of the number, like this:

```
myFloat = 32.4f;
```

Likewise, you generate a double–precision float with the double keyword:

```
double myDouble;
```

You can use the d character to force a number to double status, but doing so is unnecessary because any literal number with a decimal point is presumed to be a double.

Data Type Problems

To a human being, the integer 3 is just like the real number 3.0. However, in a computer system, the int value 3 is stored differently than the long value 3. The float and double versions of these numbers are even more different. The computer can convert between these types, but often with problems. Take a look at this segment of the Simple Math program:

```
//division by integers can cause problems
Console.WriteLine("5/4 = {0}", 5 / 4);
```

In the corresponding output, you see a surprising result.

Output:

```
5/4 = 1
```

Of course, 5 divided by 4 equals 1.25, so something went wrong. C# recognizes that both 5 and 4 are integers, so it assumes that the result of any operation between them should also be an integer. I fixed this problem by forcing 4 and 5 to be read as floating–point variables, like this:

```
Console.WriteLine("5f/4f = {0}", 5f / 4f);
```

Math Operators

Now that you know how to represent numbers, you can do basic math on them. All the basic operators work as you might expect. The plus sign (+) is used for addition, the minus sign (–) for subtraction, the forward slash (/) for division, and the asterisk (*) for multiplication.

Converting Variables

Knowing how to convert variables from one data type to another is important for programmers. C# offers many ways to perform these conversions. Sometimes, as in the Simple Math program, the conversion is done automatically. Other times, you have to do the conversion explicitly. Fortunately, C# makes variable conversion easy. As usual, it makes sense to look at a working program (see Figure 2.4):

C:\csab\cs02\convertDemo.exe
myDouble is 5.
After casting, myInt = 3.
myDouble as a String: 3.5
Please enter a number: 3.7
myString converted to double: 3.7
Please press enter to continue

Figure 2.4: Although the console works only with string values, you can convert strings to whatever

type of variable you wish.

```
using System;

namespace ConvertDemo
{

  ///<summary>
  /// demonstrates various types of variable conversions
  /// Andy Harris, 11/09/01
  ///</summary>

  class ConvertDemo
  {
    static void Main()
    {
      int myInt;
      double myDouble;
      string myString;

      myInt = 5;

      //copying an int to a double causes no problems
      myDouble = myInt;
      Console.WriteLine("myDouble is {0}.", myDouble);

      //copying a double to an int won't work!
      myDouble = 3.5;
      //myInt = myDouble;    //this line causes an error
      //Console.WriteLine(myInt);

      //You can explicitly cast, but you might lose data
      myInt = (int)myDouble;
      Console.WriteLine("After casting, myInt = {0}.", myInt);

      myString = myDouble.ToString();
      Console.WriteLine("myDouble as a String: {0}", myString);

      Console.Write("Please enter a number: ");
      myString = Console.ReadLine();

      Console.WriteLine("myString converted to double: {0}",
Convert.ToDouble(myString));
    } // end main
  } // end class
} // end namespace
```

Trick Some programmers prefer some alternate syntaxes, such as myFloat.Parse(someString) or myInt.Parse(someString). However, the Convert syntax works on all variables types, so it's a pretty convenient solution.

To illustrate some key conversion ideas, I created an int, a double, and a string. These are, by far, the most common variable types you will use. I started by assigning the value 5 (an integer value) to myInt. Then, I copied the value of myInt to the myDouble variable. This caused no problems because a double can easily hold all the information in an int. However, the next few lines of code assign the value 3.5 to myDouble and then try to copy the value of myDouble to myInt. In the preceding source code, I've placed the comment characters before this line so it will not execute. This is commonly called *commenting out* code.

I commented out that particular line of code because it causes the system to crash. You cannot copy a double value to an int because doubles have more information (namely, the information after the decimal point). You can't even copy the value 3.0 to an int variable directly because 3.0 is a

double value.

Explicit Casting

You can convert a double value to an integer value, but you will lose some information along the way. Take a look at this line:

```
myInt = (int)myDouble;
```

This line copies the value of myDouble to myInt successfully, but it does so by utilizing a trick called *casting,* in which the term (int) tells the compiler to convert the value immediately following into an integer value. This results in a loss of data, but it works. You can use this type of operation to convert any numeric data types.

The Convert Object

Strings are not like other types of data because the amount of information necessary to store a string can vary, based on the length of the string. You don't have to worry about this, but you should know that the conversion techniques that work between numbers do not work the same way with string variables. The string value "123" is *not* the same as the int value 123 or the double value 123.0. You cannot use a casting operation to convert to or from a string value. Therefore,

```
myInt = (int)"123"
```

does not work, and

```
myString = (string) 123
```

does not work, either.

In the Real World

Converting numeric data to strings is important because all the user generally sees is text information. When the user types something, the program usually sees it as a string value. For example, the Console.ReadLine() method always results in a string value. If you want to get a number from the user, you have to take the string from the ReadLine() method and convert it to a number yourself.

Fortunately, C# has an object that specializes in converting variables between types, and it always works. The convert object, pictured in Figure 2.5, is a special object that provides several useful methods. *Methods* are actions an object can perform. Most of the convert object's methods convert data from one type to another.

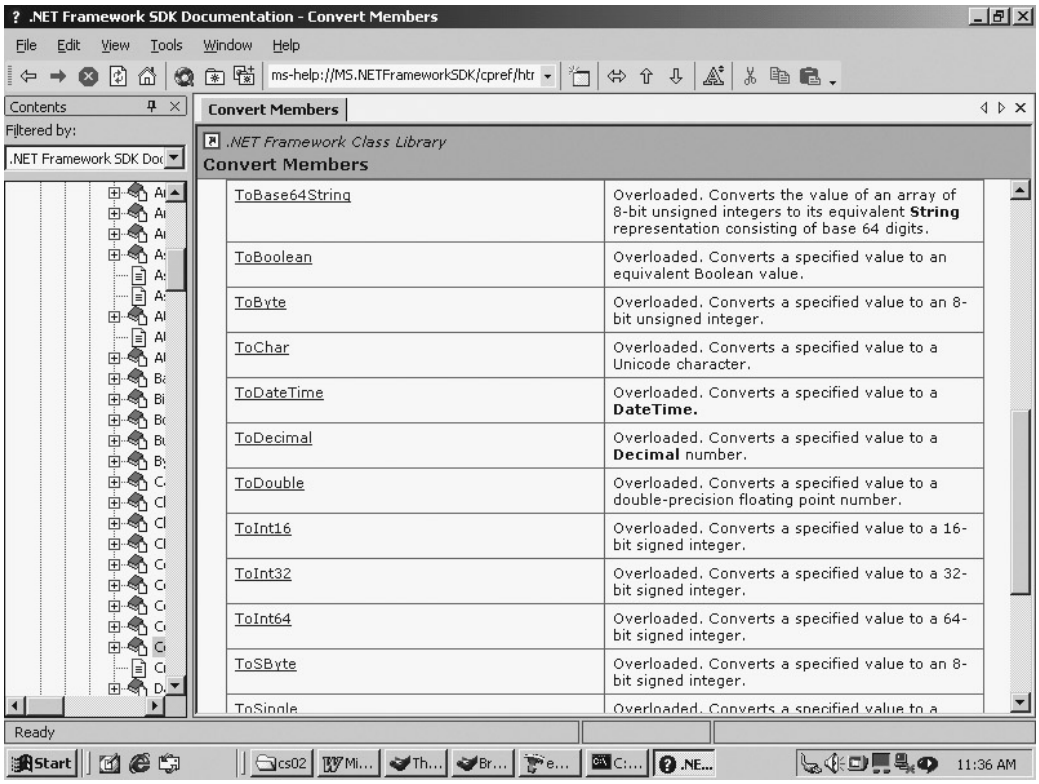


Figure 2.5: The convert object can convert nearly any variable type to any other variable type.

The Convert class is part of the System namespace, so you already have access to it. To use the class, you call one of its methods. For example,

```
myInt = Convert.ToInt32("123");
```

converts the string value "123" into an integer and stores that value into myInt. Likewise, the statement

```
myString = Convert.ToString(123);
```

takes the int value 123 and converts it into a string for storage in the myString variable. Of course, you usually won't use the Convert class with literal values, as I've done in these examples. Generally, you stuff a variable in the ToInt32() or ToString() method, like this:

```
myInt = Convert.ToInt32(myString);
```

Because you sometimes grab a numeric value from the console, you often simply put the Console.ReadLine() method inside the ToInt() parentheses, like this:

```
myInt = Convert.ToInt32(Console.ReadLine());
```

Trap The convert object isn't foolproof. If you let the user type in data, you don't have any way to know whether he or she added a decimal point. You might be tempted to write code like this: myInt = Convert.ToInt32(Console.ReadLine()); This code works great if the user types an integer, but if the user enters a real number with a decimal point, your program will crash. It is safer to do the same operation in two steps, like this: myDouble = Convert.ToDouble(Console.ReadLine()); myInt = (int) myDouble; The compiler will have no problem converting a

decimal value into a double, and then you can explicitly cast the double to an int.

Creating a Branch in Program Logic

One reason computer programs are interesting is that they can appear to change behavior. The key to this flexibility is the *condition* structure. Before I launch into a definition of the condition structure, take a look at the Hi Bill game.

The Hi Bill Game

To illustrate the point of conditions—and their cousin, the if statement—I wrote a program that checks whether the user is Bill Gates. (I'm sure that Bill is hanging out at the bookstore as I write this, eagerly waiting to buy this book.) Just in case, I'll be ready. Here's the source code for the Hi Bill program:

```
using System;

namespace AreYouBill
{
  /// <summary>
  /// Demonstrates the If Statement
  /// </summary>
  class AreYouBill
  {
    static void Main(string[] args)
    {
      string fullName;

      Console.Write("Please enter your full name: ");
      fullName = Console.ReadLine();

      //basic if statement
      if (fullName == "Bill Gates")
      {
        Console.WriteLine("Nice job on C#, Bill.");
      } // end if

      //if - else - statement
      if (fullName == "Bill Gates")
      {
        Console.WriteLine("C# is pretty cool");
      } else {
        Console.WriteLine("Sorry, I was looking for Bill");
      } // end if

      //if - else if structure
      if (fullName == "Bill Gates")
      {
        Console.WriteLine("C# is pretty cool");
      } else if (fullName == "James Gosling"){
        Console.WriteLine("Java is pretty cool");
      } else {
        Console.WriteLine("Nice to see you, {0}!", fullName);
      } // end if

      //hold for user response
      Console.WriteLine();
      Console.WriteLine();
      Console.WriteLine("Please press enter key to continue");
```

```
      Console.ReadLine();

    } // end main
  } // end class
} // end namespace
```

The program does exactly what you expect: It asks the user for a name and responds appropriately, as you can see in Figures 2.6, 2.7, and 2.8.

C:\csab\cs02\HiBill\bin\Debug\HiBill.exe
Please enter your full name: Bill Gates
Nice job on C#, Bill.
C# is pretty cool
C# is pretty cool
Please press "enter" to continue

Figure 2.6: Apparently, Bill Gates *has* been here!

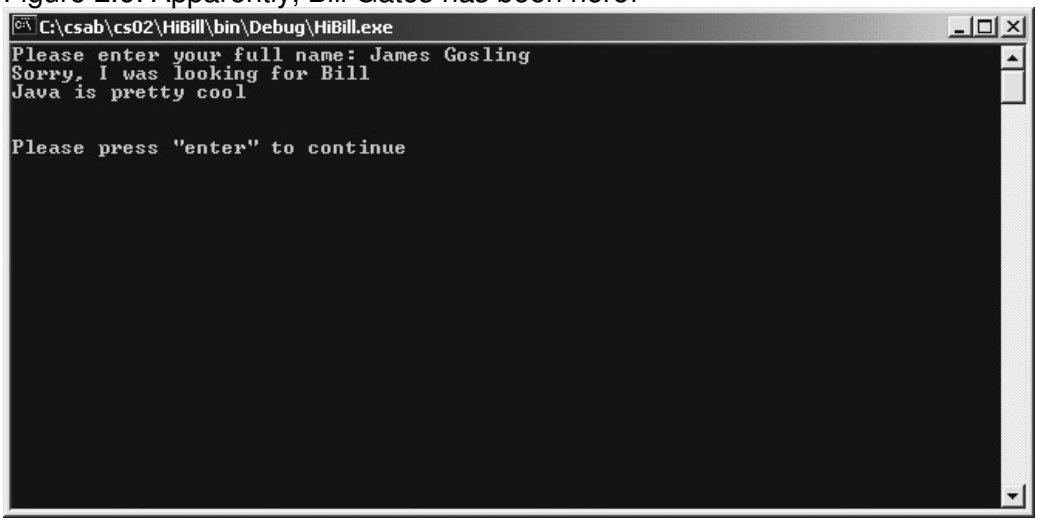
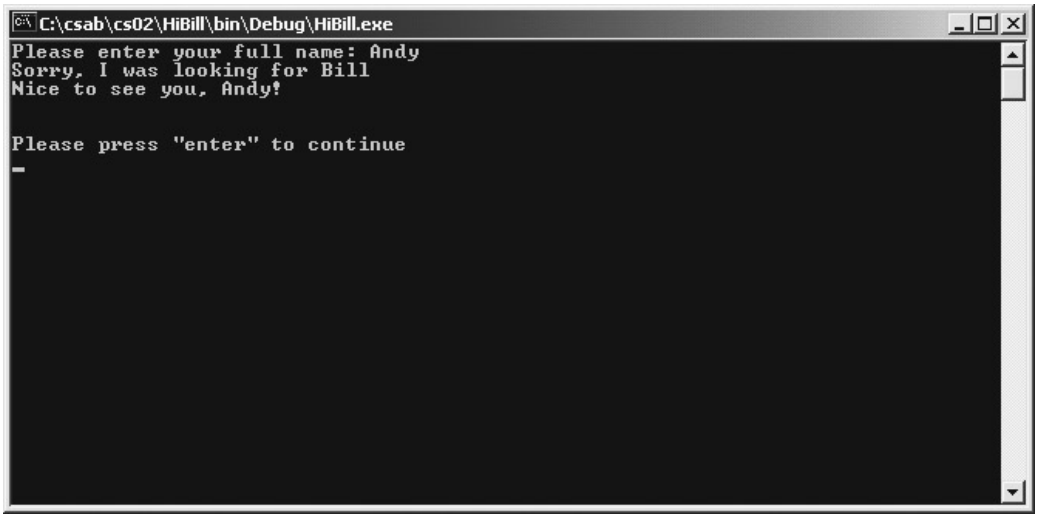

C:\csab\cs02\HiBill\bin\Debug\HiBill.exe
Please enter your full name: James Gosling
Sorry, I was looking for Bill
Java is pretty cool
Please press "enter" to continue

Figure 2.7: If James Gosling (the primary developer of the Java language) shows up, the program can respond appropriately.


C:\csab\cs02\HiBill\bin\Debug\HiBill.exe
Please enter your full name: Andy
Sorry, I was looking for Bill
Nice to see you, Andy!
Please press "enter" to continue
Figure 2.8: If users other than Bill Gates or James Gosling play this game, the program personally greets them.

Condition Testing

The key to a computer's decision–making capability is the condition. A *condition* is an expression that can be evaluated as true or false. In C#, conditions are always surrounded by parentheses, and they usually compare a value to a variable. In the Hi Bill program, the first condition looks like this:

```
(fullName =="Bill Gates")
```

The term fullName is the name of a variable. The two equal signs (==) check for equality. This condition checks whether the variable fullName is equal to the value "Bill Gates". Table 2.2 illustrates the various kinds of operators that can be used inside conditional statements.

Table 2.2: Comparison Operators

Operator	Meaning	Sample Condition
<	Is less than	(x < 5)
>	Is greater than	(x > 5)
==	Is equal to	(x == 5)
<=	Is less than or equal to	(x <= 5)
>=	Is greater than or equal to	(x >= 5)
!=	Is not equal to	(x != 5)

Please notice that the equality operator is *two* quote signs, not one. All the comparison operators can be used with any numeric or string values. If you use greater than or less than operators on string values, the computer compares the values in alphabetical order. In other words, ("Apple" < "Zebra") evaluates to true because Apple falls earlier than Zebra in alphabetical order. In essence, Apple is less than Zebra.

Trap When assigning a value to a variable, use one equal sign (=). When comparing a variable to another variable or a value, use two equal signs (==). Many programmers forget this and use a single equal sign when they should use two. C# does not compile but often gives you a strange error, such as "cannot implicitly convert type 'string' to 'bool'." (Other languages, such as C, do compile and subsequently cause problems with the program that are hard to track down.) If your program is not working, make sure that you are using the comparison operator

(==) inside your conditions.

The If Statement

The most common place to use a conditional structure is inside an if statement. The if statement is basic. Here's the simplest if statement in the Hi Bill program:

```
//basic if statement
if (fullName == "Bill Gates")
{
  Console.WriteLine("Nice job on C#, Bill.");
} // end if
```

The code starts with the keyword if, followed by a condition. In this case, the condition checks whether the fullName variable (entered by the user) is equal to the value "Bill Gates". After the condition, you see a line of code (that writes out a message, in this case) between a pair of braces({}). If the condition is true, all the statements inside the braces are executed. If the condition is false, the computer skips all the instructions inside the braces and move on to the next instruction after the right brace (}). You can have as many lines of code as you like between the braces.

The Else Clause

Sometimes you want the computer to do one thing if the condition is true and something else if the condition is false. For example, in the Hi Bill program, you want the computer to say one thing if Bill Gates is the user but something else if the user is not Bill.

In these circumstances, you can use a special addition to the if statement: the else clause. Take a look at the following segment of the Hi Bill program to see how the else clause works:

```
//if - else - statement
if (fullName == "Bill Gates")
{
  Console.WriteLine("C# is pretty cool");
} else {
  Console.WriteLine("Sorry, I was looking for Bill");
} // end if
```

You can see that the code starts out the same way as the simple if statement, but after the first right brace (}), I added the else clause and another left brace ({). All the code between the condition and the else statement will be executed if the condition is evaluated to true. The code between else and the last right brace will execute if the condition is false.

Multiple Conditions

If you are making a more complex comparison, you can also check for multiple conditions. The following code fragment checks for Bill Gates or James Gosling (the author of the Java programming language):

```
//if - else if structure
      if (fullName == "Bill Gates")
      {
        Console.WriteLine("C# is pretty cool");
      } else if (fullName == "James Gosling"){
        Console.WriteLine("Java is pretty cool");
      } else {
```

```
        Console.WriteLine("Nice to see you, {0}!", fullName);
      } // end if
```

After the keyword else, you can place another if statement to check for another condition. In this type of structure, the program checks the first condition. If it's true, the program executes the code after that condition and proceeds to the next line after the end of the entire if structure. If the initial condition is false, the program checks each succeeding condition. If none of the other conditions are true, the program executes the code following the else clause. You can use as many else if structures as you like, as long as you are careful to end each one with a right brace.

Trick Putting in a plain old else clause (without a condition) is a good idea even if you don't think you will need it. The else clause is a great place to trap any unforeseen situations and respond to them.

Working with The Switch Statement

It's relatively common to come across a situation in which you want to check one variable for a number of possible values. You can use the if...else if structure for these situations, but C# supplies a handy structure that specializes in these kinds of scenarios: the switch statement. In such situations, you use the switch statement, which I'll explain a little later, but for now take a look at the Switch Demo program, which will ease you into switch statements.

The Switch Demo Program

Look at the following source code for an illustration of the switch statement:

```
using System;

namespace SwitchDemo
{
  /// <summary>
  /// Demonstrates use of the Switch structure
  /// Andy Harris, 11/10/01
  /// </summary>
  class SwitchDemo
  {
    static void Main(string[] args)
    {
      string fullName;
      string greeting;

      //get name from user
      Console.Write("Please Enter your full name: ");
      fullName = Console.ReadLine();

      //check name
      switch (fullName){
        case "Bill Gates":
          greeting = "Great job on C#";
          break;
        case "James Gosling":
          greeting = "That Java thing is really cool";
          break;
        case "Alan Turing":
          greeting = "The Turing machine was pretty amazing";
          break;
```

```
        case "Grace Hopper":
          greeting = "Wow. You discovered the first computer bug!";
          break;
        default:
          greeting = "We're waiting for your contribution to computer science,
" + fullName;
          break;
      } // end switch

      //write response
      Console.WriteLine(greeting);
      Console.WriteLine();
      Console.WriteLine();
      Console.WriteLine("Press \"enter\" to continue");
      Console.ReadLine();
    } // end main
  } // end class
} // end namespace
```

Examining How Switch Statements Work

The switch statement looks at one variable or expression (in this case, the variable fullName) and compares it to several cases. In essence, this code

```
switch (fullName){
        case "Bill Gates":
          greeting = "Great job on C#";
          break;
```

is equivalent to the following code, which contains a switch statement:

```
if (fullName == "Bill Gates")
{
  greeting = "Great job on C#";
}
```

The variable you are comparing belongs in a pair of parentheses right after the keyword switch. The rest of the structure goes inside a pair of braces. For each value you want to compare, you build a case structure. This structure describes the value you are comparing the variable to. For example, "Bill Gates" is one possible value of userName, and "James Gosling" is another, so each of these terms makes up a case. The case structure begins with the keyword case, followed by the value you want to compare and then a colon (:) character. You must end each case with a break statement, which informs the computer that you are finished considering this possible value for the expression. The break structure helps the computer understand that you are done writing code that should happen if the user is Bill Gates, for example, and you're ready to start the next case (which might be James Gosling).

Trap If you are familiar with another programming language, take a careful look at the switch statement. In C#, the switch statement differs from its cousins in the other popular languages. It is possible to switch on a string variable (this is impossible in C), and C# *requires* the break statement at the end of each case, unlike Visual Basic or C.

The switch statement has a section named default:, which acts like the else clause in an if structure. If none of the other cases turn out to be true, the code in the else clause executes. It's a good idea to include a default clause in any switch statement you build.

Trick

If you want to see what's going on in your code, you can step through it one line at a time. While you are in the IDE (*Integrated Debugging Environment*), press F11 to run one line of code. You will see the current line of code highlighted in yellow. Keep pressing the F11 key to see how the computer walks through your code and how it skips over elements. This is a great way to see how branching structures work.

Creating a Random Number

In most game situations, the computer plays the opponent or simulates an unpredictable situation. The key to game programming is the idea of random number generation. You can ask the computer to come up with a random number, which you can use to determine how the computer should proceed.

In the Real World

Random number generation is not only for games! You can also use this technique any time you want to simulate uncertainty. Business simulations and testing programs (software designed to test other software) are two types of applications that rely heavily on random number generation. Random numbers also play a key role in a class of programs called *neural nets* which can often provide answers to otherwise unsolvable problems.

Most programming languages have some sort of built−in random number generation routine. Usually, this is a command or function that returns back a random double value between 0 and 1.

Trap You should know that the numbers aren't truly random. Instead, they are generated by a complex formula that tries to create numbers as close to purely random as possible. The results are close enough for the purposes of this book.

C# uses a unique but powerful approach to random number generation. It provides a special object named the (surprise!) Random class. This class has the capability to create several kinds of random values, but you will focus on extracting a double value because you can create any other kind you might need from a double.

Introducing the Die Roller

As usual, I'll show you a program to illustrate what I mean. The DieRoller program featured in Figure 2.9 illustrates a common problem. I want the computer to simulate rolling a six−sided die.

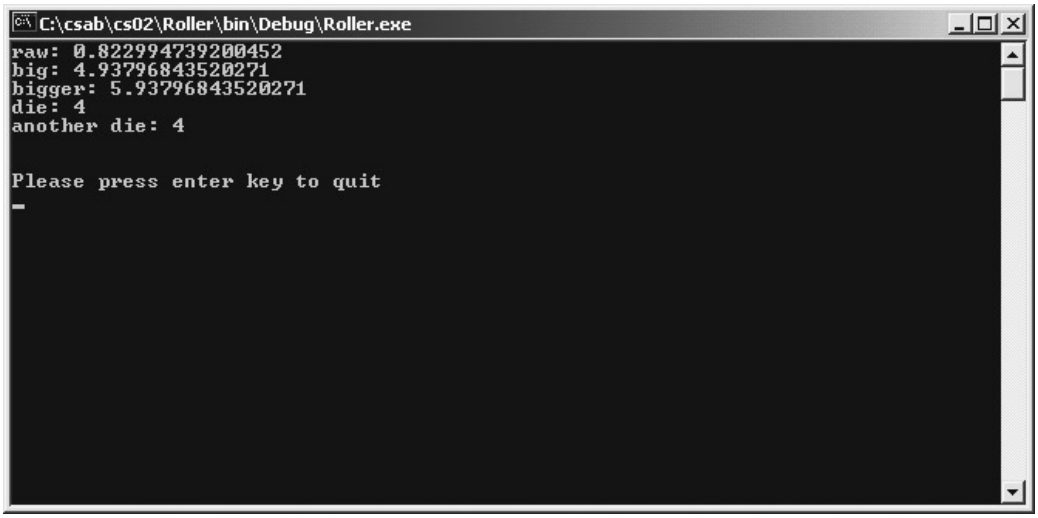


Figure 2.9: The program “rolls up” two six–sided dice.
The random object returns a decimal value between 0 and 1, but a standard die has integer values from 1 to 6. The program illustrates how you can go from a 0–1 value to whatever kind of random number you want.

Exploring the Random Object

The random object is different from the objects you have seen so far (the console object and the convert object). In those cases, the objects were already available simply because you are in the System namespace. The random object is also available in the System namespace, but you can’t use it directly. Instead, it must be created as a variable in order to use it. To create a random object named generator (that seems like a good name to me), I used this line of code inside the Main() method:

```
Random generator = new Random();
```

The generator is a variable, but it isn’t a string or integer you’ve seen before. Instead, it is a reference to a random object. It is created much like any other variable, but you use the new keyword to specify that the computer will be making an object instead of a simple variable. It’s okay if this confuses you right now. It will make much more sense when you start making objects of your own in Chapter 4, "Objects and Encapsulation: The Critter Program."

Trick Remember, I learned about the random object and its methods by digging around in the online help and the .NET documentation. These are always good ways to learn about new objects that can help you solve problems.

Creating a Random Double with the .NextDouble() Method

After you set up the generator, you can get a double value easily. Use the NextDouble() method to get a double value from the generator. Here’s some code that will create a random object named generator, get a random double value from it, and store that value in a variable named myDouble:

```
double myDouble;
Random generator = new Random();
myDouble = generator.NextDouble();
Console.WriteLine(myDouble);
```

The resulting value will be a number from 0 to 1 with a lot of decimal values. This number is nice,

but the goal is to simulate a die, which has a value from 1 to 6.

Getting the Values of Dice

The basic problem of random number generation in computer programs is that computers usually create 0–1 numbers, and you almost always need them in another format. Take a look at the code for the Roller program, and you will see how I solved that problem:

```
using System;

namespace Roller
{
  /// <summary>
  /// Demonstrates creation of a random number
  /// Andy Harris, 11/10/01
  /// </summary>
  class Class1
  {
    static void Main(string[] args)
    {
      double raw;
      double big;
      double bigger;
      int die;
      Random generator = new Random();

      raw = generator.NextDouble();
      Console.WriteLine("raw: {0}", raw);

      big = raw * 6;
      Console.WriteLine("big: {0}", big);

      bigger = big + 1;
      Console.WriteLine("bigger: {0}", bigger);

      die = (int)big;
      Console.WriteLine("die: {0}", die);

      //do it all in one step
      die = (int)(generator.NextDouble() * 6) + 1;
      Console.WriteLine("another die: {0}", die);

      Console.WriteLine();
      Console.WriteLine();
      Console.WriteLine("Please press enter key to quit");
      Console.ReadLine();
    } // end main
  } // end class
} // end namespace
```

Getting the desired number is not difficult, but it takes some thought. I did it in several steps in the Roller program so that you can follow the process. First, the variable raw simply gets a value from the generator. This number will contain a 0–1 value. In the next step, I multiply raw by 6 to get a variable named big. If raw is 0 (which will almost never happen), 0 multiplied by 6 will be 0. If raw is 1, (which will almost never happen, either), raw multiplied by 6 will be 6. Most of the time (close enough to all the time), the result of raw multiplied by 6 will be larger than 0 and smaller than 6.

For example, if raw is 0.341061992729577, big will be this value multiplied by 6, or 2.04637195637746. If you convert this number to an integer, you will be close to the desired results

because you would have a random integer between 0 and 5. However, most dice are numbered 1–6. Therefore, the next step is to add 1 to big. Looking at the same example, bigger is big + 1, or 3.04637195637746. The last step is to lop off the decimal part of the number, which is easily done by casting bigger to an integer.

It might help you understand the results if you run the program a few times and watch the relationships between the numbers.

When you understand how the pattern works, you might prefer to put all the steps together. The line in the program that looks like this

```
die = (int)(generator.NextDouble() * 6) + 1;
```

does exactly that. It gets a double from the generator, multiplies its value by 6, casts the result as an integer, adds 1, and copies the result to the die variable. Either approach is acceptable, but most programmers use the second technique because it's easier to type.

Trick You can use a similar technique to simulate any kind of random number you want. If you want to duplicate the 20–sided die used in certain board games, simply replace the value 6 with 20 in the preceding expression.

Creating the Math Game

You now know everything necessary to put together the simple Math Game from the beginning of this chapter. The game itself is not complex when you know how all the pieces work.

Designing the Game

The first thing to think about is the general design of the game. This might seem like the easiest step, but often it is the most difficult and frequently overlooked. In this case, I was looking for a simple game that would illustrate the concepts of variables and branching behavior. I also wanted to generate random math problems that would be appropriate for children in elementary school. The program should present four problems, one each of addition, subtraction, multiplication, and division. The answers should always be positive integers. When players finish the quiz, they should get a numeric score and some feedback about their score.

In the Real World

It's smart to think about exactly what you want your program to do and even to write it down before you start programming. Later, you're bound to get tied up in details. When things go wrong, you will be glad to have a plan you can fall back on.

Creating the Variables

I started my program by doing the normal modifications of the default code and adding a few key variables:

```
using System;
```

```
namespace MathGame
{
  /// <summary>
  /// Simple Math Game
  /// Asks four math questions
  /// Using Random Numbers
  /// Andy Harris, 11/7/01
  /// </summary>
  class Game
  {
    static void Main(string[] args)
    {
      int a, b, c, guess, score;
      score = 0;
      //create the random number generator
      Random roller = new Random();

Console.WriteLine("Welcome to the math Game!  I'll give you some simple problems to solve.");
```

I created variables for a and b, which will be the two random numbers in each problem. I also created the variable c to hold the sum or product of a and b. The variable guess holds the user's response to questions, and score keeps track of the number of questions the user answers correctly. Note that you can create several variables with the same int statement.

Also notice that I set the value for score to start at zero.

The program will use many random numbers, so I created a random object named roller to make the numbers. Finally, I added a greeting so that the user would have some idea of what's going on.

Managing Addition

The addition problem sets the stage for all the other questions:

```
//addition
      a = (int)(roller.NextDouble() * 10) + 1;
      b = (int)(roller.NextDouble() * 10) + 1;
      c = a + b;

      Console.Write("What is {0} + {1}? ", a, b);
      guess = Convert.ToInt32(Console.ReadLine());

      if (guess == c) {
        score++;
      }
```

The program gets a value for a that is between 1 and 10. I multiplied the double value from roller by 10, added 1, and converted it to an integer. The program gets a similar value for b. The c variable is calculated by adding a and b. The program writes out the question to the screen, interpolating the values for a and b.

The next line gets a response from the console, converts it to an integer, and then sends the resulting value to the variable guess.

Finally, the program checks whether the guess is correct. If so, the value of score is incremented. Note the line that increments the score.

score++ is a special shorthand for score = score + 1. Because you frequently need to increment by 1, the ++ operator is a very handy little shortcut.

In the Real World

When developers set out to improve on the C language, they wanted to illustrate that it was better than C, so they named it C++. Because C# is supposed to be an improvement over C++, I wonder if it should be named C++++!

Managing Subtraction

You might be tempted to duplicate the addition code four times and simply change the operators from addition to the other math operations. However, this will cause some problems. Remember that you want to keep your results all positive integers. Because both a and b are random, how will you ensure that a minus b is always positive? One solution is to subtract b from a if a is bigger, or a from b if b is bigger. However, there's a more elegant solution. Look at the code and see whether you can figure it out before I explain it:

```
//subtraction
a = (int)(roller.NextDouble() * 10) + 1;
b = (int)(roller.NextDouble() * 10) + 1;
c = a + b;

Console.Write("What is {0} - {1}? ", c, a);
guess = Convert.ToInt32(Console.ReadLine());

if (guess == b) {
  score++;
```

Nothing says that the user has to be given the random values! What I did was get two random values, as before, and add them together so that c is the sum of a and b. I then asked the user what c minus a is, knowing that the response would be b.

Managing Multiplication and Division

The multiplication and division segments of the code are very much like the addition and subtraction sections. For the division problem, I multiplied a and b and asked the user what c divided by a is, knowing that, again, the answer would be b.

```
//multiplication
a = (int)(roller.NextDouble() * 10) + 1;
b = (int)(roller.NextDouble() * 10) + 1;
c = a * b;

Console.Write("What is {0} * {1}? ", a, b);
guess = Convert.ToInt32(Console.ReadLine());

if (guess == c) {
  score++;
}

//division
a = (int)(roller.NextDouble() * 10) + 1;
b = (int)(roller.NextDouble() * 10) + 1;
```

```
c = a * b;

Console.Write("What is {0} / {1}? ", c, a);
guess = Convert.ToInt32(Console.ReadLine());

if (guess == b) {
  score++;
}
```

Checking the Answers

All that remains is to analyze the score. This is easily done with a switch structure:

```
      Console.WriteLine("Score: {0} out of 4", score);

      switch (score)
      {
        case 4:
          Console.WriteLine("You're a genius!");
          break;
        case 3:
          Console.WriteLine("You're pretty smart!");
          break;
        case 2:
          Console.WriteLine("You could do better");
          break;
        case 1:
          Console.WriteLine("You could use some practice");
          break;
        case 0:
          Console.WriteLine("Maybe you were good at gym class in school");
          break;
        default:
          Console.WriteLine("Hey, something went wrong here!");
} // end switch
```

The switch statement simply checks all the possible values of score and sends an appropriate (and occasionally condescending) message to the user. Note that I added a default clause, even though it should not be possible for score to have a value besides 0, 1, 2, 3, or 4. Things can and do go wrong in programming, so if something does go wrong here, having a message on the screen to note that an error occurred is much better than having the program crash.

Waiting for the Carriage Return

I like to let the screen stick around until the user gets a chance to read it, so I added the Please press Enter key code at the end of this program, as I do in all my console programs. I also ended all the structures, watching out for indentation and comments:

```
      //hold the screen to see results
      Console.WriteLine();
      Console.WriteLine();
      Console.WriteLine("Please press enter key to quit");
      Console.ReadLine();
    } // end main
  } // end class
} // end namespace
```

Summary

This chapter leads you through the murky swamp of computer mathematics (well, around the fringes anyway). You have learned a few important ways that computers store numeric values. You've learned the basic numeric variable types and how to convert between them. You know how to create a random double and how to change that double into any range of integers you want. You also know how to cause a branch in the computer's behavior, based on a certain condition. You should be proud because you've learned a lot already. In the next chapter, you will learn how to make your programs repeat, which will contribute tremendously to the kinds of programs you can write.

Challenges

- Write a program that simulates a coin toss. Generate a random number between 0 and 1. If that value is less than .5, make it heads. Otherwise, output tails.
- Many role–playing games require dice with a different number of faces than the traditional six–sided cube. Write a program that asks the user how many sides the die should have, and give a random result in the appropriate range.
- Write a program that simulates a "loaded" die that comes up with the value 1 for half the time and some other random value the other half of the time.
- Make the Math Game more advanced by incorporating other math operators. Look up the math object in the .NET reference for interesting operators such as Math.Abs (absolute value) and Math.Pow (used for exponentiation).

Chapter 3: Loops and Strings: The Pig Latin Program

One of the more useful aspects of computer programs is their capability to repeat tasks over and over. In this chapter you will learn how to make your programs repeat parts of themselves. You will learn a couple types of looping structures and how to decide which one to use for a particular task. Also, you'll take a closer look at string variables and learn many capabilities of strings in the C# language. Finally, you'll investigate how to begin designing complex programs. After reading this chapter, you will be able to

- Examine an object in the Object Browser.
- Use the most common **string** methods.
- Make your program repeat a given number of times.
- Make a program that counts backwards and skips numbers.
- Write loops that continue an indefinite number of times.
- Avoid the most common traps when building loops.
- Use the STAIR approach to manage the planning process.

Project: The Pig Latin Program

Your goal this chapter is to write another simple and fun program. This program replicates a silly word game that is very popular with children. The user types in a phrase, and the program calculates the pig latin translation of that phrase. In case you've forgotten, pig latin uses a simple formula to make an English word sound like a Latin word: If the word begins with a vowel, you add *way* at the end of the word. If the word begins with a consonant, you move the consonant to the end of the word and add *ay*. Look at the program in Figure 3.1 to see an example.

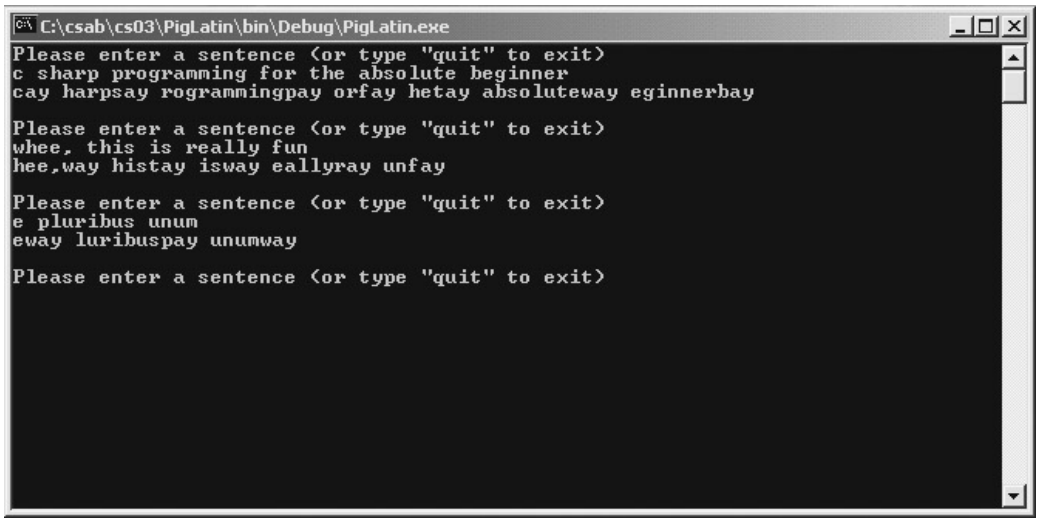

C:\csab\cs03\PigLatin\bin\Debug\PigLatin.exe
Please enter a sentence (or type "quit" to exit)
c sharp programming for the absolute beginner
cay harpsay rogrammingpay orfay hetay absoluteway eginnerbay
Please enter a sentence (or type "quit" to exit)
whee, this is really fun
hee,way histay isway eallyray unfay
Please enter a sentence (or type "quit" to exit)
e pluribus unum
eway luribuspay unumway
Please enter a sentence (or type "quit" to exit)

Figure 3.1: The title of this book sounds very classy when translated into pig latin.

The Pig Latin program utilizes a couple features that have not been covered yet in this book, such as text manipulation and repetition. Obviously, the program uses techniques for manipulating text. C# supplies many interesting ways to work with text values, which you will learn about in the next section. Also note that the program repeats in two ways. First, it continues to prompt for a new phrase until the user types in *quit*. Also, it does the appropriate manipulation to each word in the phrase. Somehow, the program knows how many words are in the phrase. Read on, and you will learn about this and a little more.

Investigating The String Object

The Pig Latin program manipulates text. As you recall from Chapter 1, “Basic Input and Output: A Mini Adventure,” most programmers, including those who program in C#, refer to text variables as *strings*. C# provides a special object called the *String* which manipulates text values in a number of ways. After you understand how a String object works and how it relates to text, you can capitalize the text, search for one phrase inside another, find out the length of the text, and many other things.

The String Mangler Program

The String Mangler program is a silly program that demonstrates ways you can fold, spindle, and mutilate an innocent string variable. I started by defining a string variable and then did some interesting things with it.

Take a look at the program in Figure 3.2 to see what you can do with strings in C#.

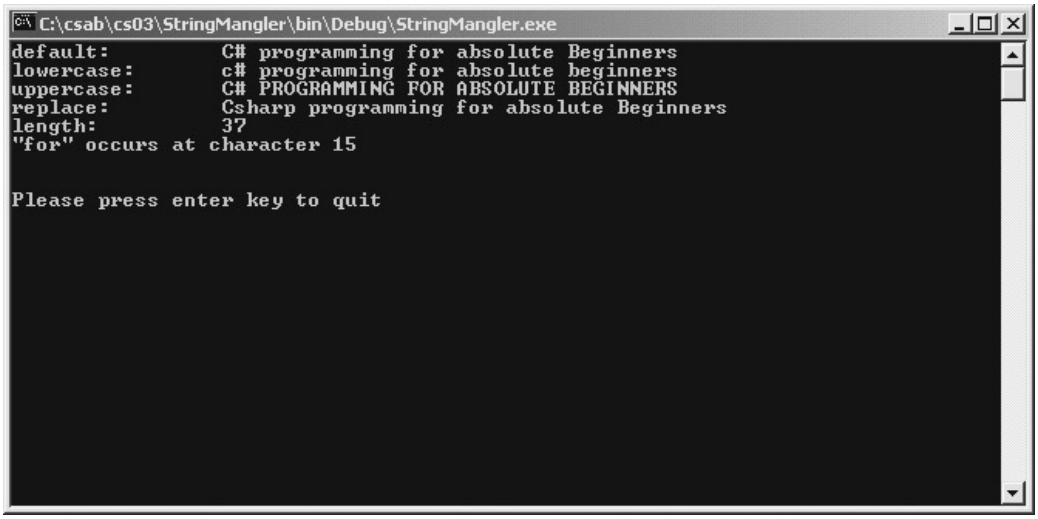

C:\csab\cs03\StringMangler\bin\Debug\StringMangler.exe
default: C# programming for absolute Beginners
lowercase: c# programming for absolute beginners
uppercase: C# PROGRAMMING FOR ABSOLUTE BEGINNERS
replace: Csharp programming for absolute Beginners
length: 37
"for" occurs at character 15
Please press enter key to quit

Figure 3.2: You can do many interesting things with string variables, including converting to upper and lower case, searching for a phrase, and determining the length of a phrase.

I started with a string that contains the title of this book. (Catchy, huh?) I then did a few simple manipulations of the string. First, I printed it out without any changes. On the next line, I converted it entirely to lowercase and then entirely to uppercase. Next, I replaced the pound sign (#) with the word *sharp*, and I figured out where the word *for* occurs in the title of the book. You can do a lot more with string values, but these examples illustrate the possibilities.

A Closer Look at Strings

C# is an object–oriented programming language. You will learn the implications of object–oriented programming as you progress through this book. One implication of object–oriented programming in C# is that you encounter mostly objects. In the first chapter you learned about string variables. The term *string* is just the programmer’s way of saying *text*. Most languages have a special type of variable for handling strings. In C#, a string variable is actually an object, which

is a little more powerful than a normal variable. Regular variables contain only data, but objects can have data (like the text in a String object) and commands for manipulating the data. The String object has several very important properties and methods. Programming in C# is focused on learning about the various objects the language provides to you. In particular, if you want to manipulate text, you’ll need to know how to use the String object.

Hint Although I am showing you how to learn more about the String object in this section, the real lesson is much broader than that. *All* the interesting variables and commands in C# are related to objects, so one key to becoming a good C# programmer is learning how to investigate the various objects you encounter. Fortunately, the IDE (*Integrated Debugging Environment*) provides powerful tools for learning about the objects in the .NET system. After you learn use these tools (such as the Object Browser, which I discuss next), you will find your path to C# proficiency reasonably straight.

Using the Object Browser

In the previous chapters I showed you how to use the .NET SDK documentation and the online help within the editor to learn about objects in C#. However, there is another method, and most programmers find it more convenient. Figure 3.3 demonstrates the Object Browser, an important tool of the Visual Studio environment.

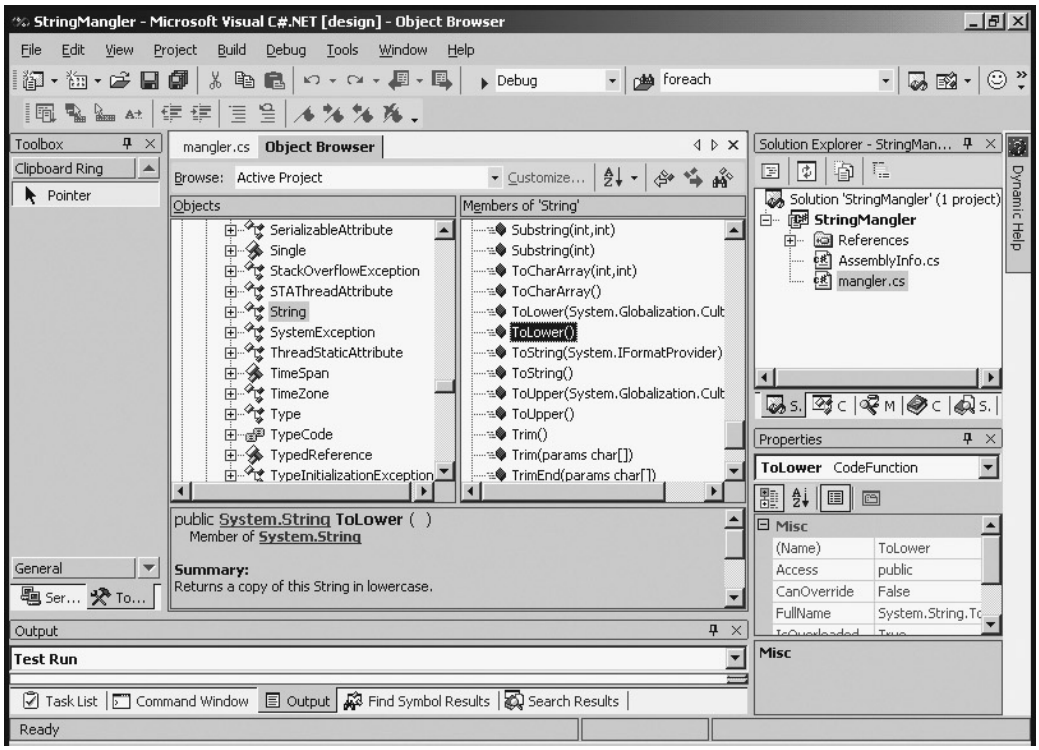


Figure 3.3: The Object Browser enables you to get online help on objects quickly.

You can reach the Object Browser in the IDE by selecting View, Other Windows or pressing Ctrl+Alt+J. All the objects available to your program are accessible from this tool. You find the .NET objects (such as string) under the mscorlib (Microsoft Core Library) branch of the tree. This is usually the first element visible in the Object Browser. Again, understanding the relationship between namespaces and objects is critical. The mscorlib is the library of all .NET objects. Open this library by clicking on its name, and you see a list of the namespaces. Find the System namespace, and you see a list of objects in the namespace.

The String object is an element of the System namespace. When you click the word *String* in the Objects tree, you see a list in the Members panel on the right. This shows the characteristics, or *properties*, of the String object.

The string object also has methods. Properties describe an object, and methods are actions the object can take. Put another way, properties are adjectives, and methods are verbs related to a specific object.

Take a careful look at the properties and methods of the string object. The String Mangler program uses these string characteristics to do its magic.

In the Real World

Understanding C# might be easier if you envision a more concrete object. OOP programmers create a world populated by various types of objects. Shortly, you will start to build objects of your own. Thinking ahead to that process might help you understand how properties and methods work. If you wanted to build a **cow** object, for example (really, I've done it, in an odd set of circumstances), you would start by thinking about a cow's properties (**age**, **breed**, **gender**) and its methods (**giveMilk**, **Moo**, **chewCud**). The **string** object isn't quite as fun to think about as the **cow** object, but it works the same way. The designers of C# thought about the characteristics a **string** should have and made those into properties. They also thought about what a **string** should do and made those actions into methods. Understanding this is important because you frequently use premade objects in C#. Soon enough, you'll be making your own objects, and your objects will have properties and methods. It will make a little more sense after you have made a few objects of your own starting in the next chapter.

Experimenting with String Methods

Take a look at the source code of the String Mangler program, and you'll see how the String Mangler works its magic:

```
using System;

namespace StringMangler
{
  /// <summary>
  /// Demonstrates some of the methods of the String object
  /// by Andy Harris, 11/16/01
  /// </summary>
  class mangler
  {
    static void Main(string[] args)
    {
      string theString = "C# Programming for Absolute Beginners";
      Console.WriteLine("default: \t {0}", theString);
      Console.WriteLine("lowercase: \t {0}", theString.ToLower());
      Console.WriteLine("uppercase: \t {0}", theString.ToUpper());
      Console.WriteLine("replace: \t {0}", theString.Replace("#", "sharp"));
      Console.WriteLine("length: \t {0}", theString.Length);
      Console.WriteLine("\"for\" occurs at character {0}", theString.IndexOf("for"));

      Console.WriteLine();
      Console.WriteLine();
      Console.WriteLine("Please press enter key to quit");
      Console.ReadLine();

    } // end main
  } // end class
} // end namespace
```

The first part of the program simply creates a string variable named theString and prints it to the screen in the normal way. The next line prints a modified version of the string to the screen. Here is the only part of the code that is new:

```
theString.ToLower();
```

Remember that theString is a string variable. Because it's also a string object, it has access to all the methods and properties of a string from the .NET library. Therefore, theString.ToLower() converts theString to lowercase. Of course, you probably guessed that. One benefit of object–oriented programming is increased ease of reading. The ToLower() method converts a string to lowercase. To be sure, take a look at the Object Browser for the string object, and look at the ToLower() method. (There are two versions of this method, but ignore the one that mentions Globalization.CultureInfo.) You can see a concise definition of this method, giving you a good hint about what the method does. If you need more information, you can always go to the online help.

In the Real World

You might wonder why I mention the Object Browser at all if everything in it is more completely described in the online help. I do so because the complete help system for .NET is massive, and almost nobody installs the entire thing on his or her own computer. It's common to be without the MSDN CDs or Web access at a critical point. The information in the Object Browser is always available, even if you're on somebody else's machine without the online help installed. Besides, when you know what you're looking for, the information on the Object Browser is usually enough to get you started.

Performing Common String Manipulations

The ToUpper() and ToLower() methods are used to change the case of a string, which is especially useful when you want to compare two strings. C# is very picky about case when comparing string values; therefore, "whoo hoo" is not considered the same as "WHOO HOO". If you want to check for a string input, but you don't care what case it was written in, you can write code like this:

```
Console.Write ("Please enter an exclamation");
theString = Console.ReadLine();
if (theString.ToUpper() == "WHOO HOO"){
  Console.WriteLine("That's a great saying!");
} // end if
```

Trap If you are using the ToUpper() method on a value you're comparing, make sure that you compare it to a string that's also entirely uppercase. If the condition in the preceding code fragment looked like this, if (theString.ToUpper() == "Whoo Hoo"){ the condition would always evaluate to false because any string that is converted to uppercase will *never* match a string with lowercase characters in it.

The String Mangler program illustrates some other interesting string manipulations. The replace() method is used to replace one value with another. I used it to replace the sharp sign (#) with the word *sharp.* This feature is handy if you are writing a program that automatically manipulates text files, for example. The length property returns how many characters the string has. This is especially useful if you want to look at the phrase one character at a time.

Finally, the indexOf() method gives you the ability to search for one string inside another. If the search string is not found, the method returns a –1. If the search string is located, the method returns the position in the string where the search string is found.

Trap If you look carefully at the output, you might be surprised by the result of the indexOf() method. It indicates that *for* occurs at character 15 of *C# Programming for*

Absolute Beginners. However, if you count the characters, you find that the word *for* starts at character 16! The reason for this anomaly is that humans usually begin counting with the number 1. Computers almost always begin counting with the value 0. This can trip you up if you're not careful.

You can do much more with strings, and there are variations of the methods I have shown you. However, the real focus here is not to show you every method of the string object. Instead, I hope that you will see how you can investigate the string object (or any other object you might encounter) so that you can exploit its properties, methods, and events.

Using a For Loop

The branching behavior you learned in Chapter 2, "Branching and Operators: The Math Game," is very important because it gives your programs the capability to make rudimentary choices. The other major way to control the flow of your programs is through looping behavior. *Loops* are code structures that allow parts of your program to repeat. There are a couple standard types of loops. One prominent type of looping structure repeats a code segment some specified number of times. This counting loop is called the for loop. To demonstrate the for loop, I'll imitate a bureaucrat.

Examining The Bean Counter Program

Although many of my sample programs are pointless, this one takes special pride in looking as though it's doing something important. The program featured in Figure 3.4 simply counts beans.

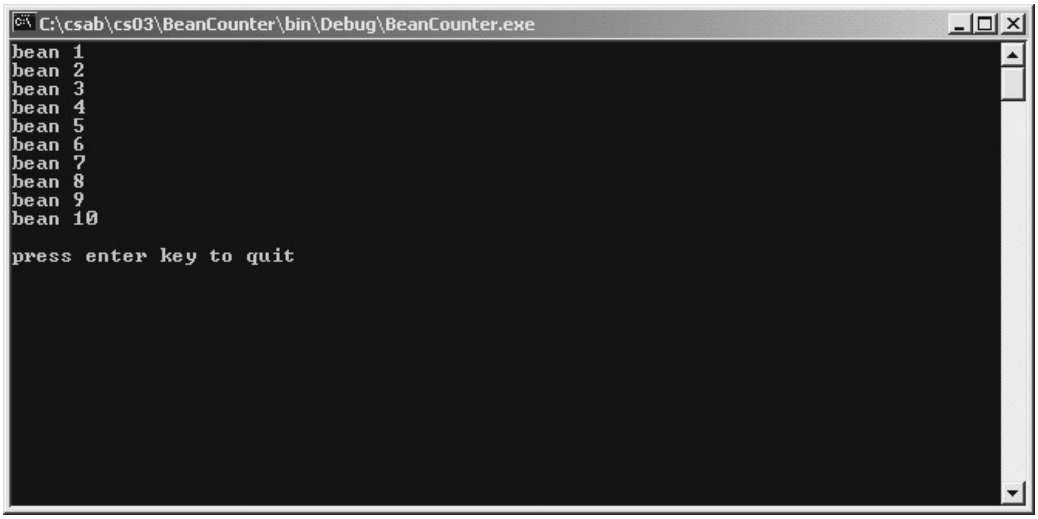


Figure 3.4: The bean counter uses a for loop to repeat behavior.

The program behind this code is reminiscent of a bureaucrat trying to look busier than he is. If you examine the output, you would expect there to be 11 different WriteLine calls, but when you examine the code, you see that there are only two! The for loop structure causes the two WriteLine calls to repeat.

```
using System;

namespace BeanCounter
{
  /// <summary>
  /// Repeats a simple task a number of times
  /// Demonstrates the basic for loop
  /// Andy Harris, 11/29/01
```

```
  /// </summary>
  class Counter
  {
    static void Main(string[] args)
    {
      int beanNumber;
      for (beanNumber = 1; beanNumber <= 10; beanNumber++){
        Console.WriteLine("bean {0}", beanNumber);
      } // end for

      Console.WriteLine();
      Console.WriteLine("press enter key to quit");
      Console.ReadLine();

    } // end main
  } // end class
} // end namespace
```

The only new part of this program is the line starting with for, which indicates the beginning of a for loop. Only three elements belong in the parentheses after for. These elements help to ensure that the loop will operate smoothly. The for line is followed by a set of braces containing one or more lines of code. The code inside the braces will repeat a certain number of times, based on the way the for loop is set up.

Creating a Sentry Variable

The part that says beanNumber = 1; establishes a special variable that will be used to control the loop. The value of this variable will control how long the loop continues. Because the variable is like a gatekeeper for the loop, it is frequently referred to as a *sentry variable*. For loops almost always use integers as sentry variables. It is important that I established the starting value of the variable at 1. As you will see shortly, you can start with other values, but most often your loop's sentry variable will start at 0 or 1.

Checking for an Upper Limit

The next part of the for loop is a condition that checks whether the variable is past a limit. In the Bean Counter program, this section looks like beanNumber <= 10;. As long as the condition is true (in this case, as long as beanNumber is less than or equal to the value 10), the loop will continue. As soon as the condition is evaluated to false (because, for example, beanNumber is 11), the loop will stop, and the next line of code after the right brace (}) will execute.

Incrementing the Variable

The last part of the for line increments the sentry variable. Remember from Chapter 2 that the beanNumber++ statement is actually a shortcut for beanNumber = beanNumber + 1. The sentry variable must be incremented for the loop to exit.

Examining the Behavior of the For Loop

Even though I have explained the structure of the for loop statement, I recommend that you take advantage of the IDE's terrific debugging mode so you will really understand what's going on. Load or type the Bean Counter program in your editor, and use the F11 (function key F11) to run your program one line at a time. As you run your program, you will see the current line in your editor highlighted in yellow. Be sure that the debugging box at the bottom–left of the screen is set to Autos

(as I have done in Figure 3.5), and you'll see the value of beanNumber, which starts at 0.

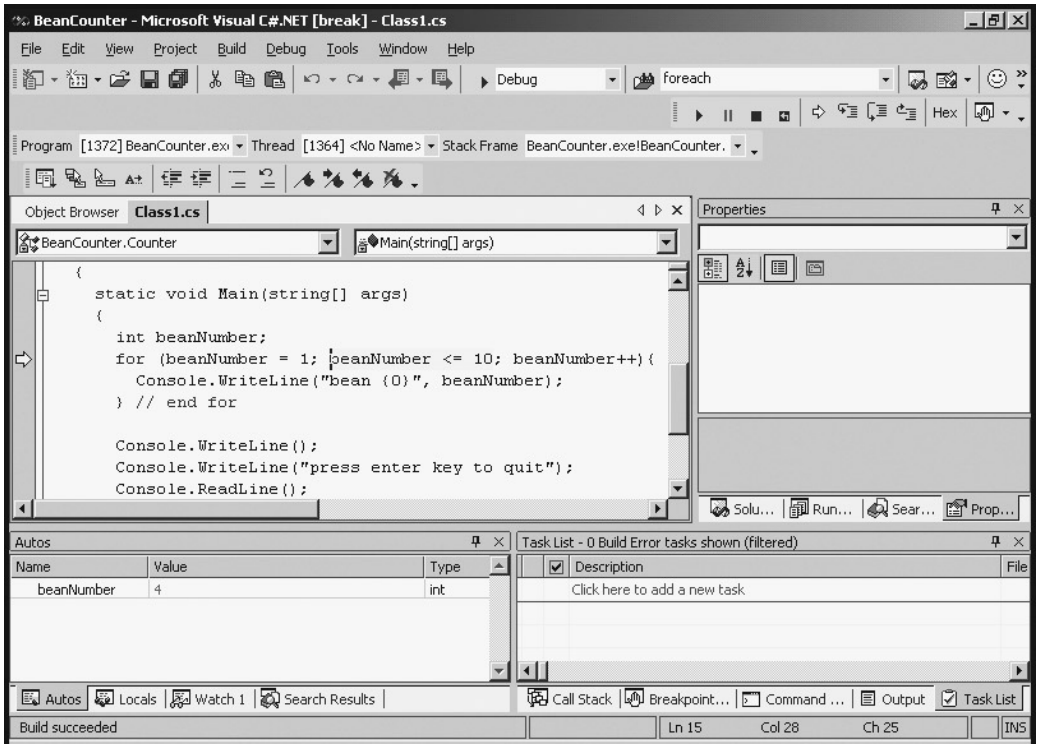


Figure 3.5: The highlight indicates the current line being executed, and the Autos window describes the value of the variables.

Press the F11 key repeatedly, and see what happens to the highlight indicating program control. Also watch what happens to the beanNumber variable. The first time through, the program sets the value of beanNumber to 1 and then checks whether beanNumber is less than or equal to 10. It is, so the code in the loop activates. After the WriteLine code happens, look at the output screen. (It is probably running, but you will not see it unless you call it to the foreground by clicking its icon in the task bar.) When the program reaches the right brace that ends the for loop, it goes back to the top of the loop. It increments the variable and checks the condition again. It still finds the condition true, so it continues executing the loop. Eventually, the value of beanNumber will be larger than 10, so the loop will exit.

Trick Learning how and when to use the debugging mode can save you hours of frustration. The worst kinds of bugs to fix are those that occur when the program is working *almost* correctly, but somewhere something is going wrong. The ability to step through the code one line at a time and examine the values of the variables in slow motion is invaluable. It's also a great technique for making sure that you understand how program control is flowing, as you're doing in this example.

Varying the For Loop's Behavior

Several variations of this basic loop are used for different kinds of counting situations. You can set up for loops that skip over numbers, count by two, or count backwards. The basic design of the for statement is the same, but you can change the way the three parts of the for loop are written to change the way the loop acts.

The Fancy Beans Program

The Fancy Beans program demonstrates a few variations on the theme of the basic for loop. Take a look at the output of this program in Figure 3.6, then I'll show you how the code works.

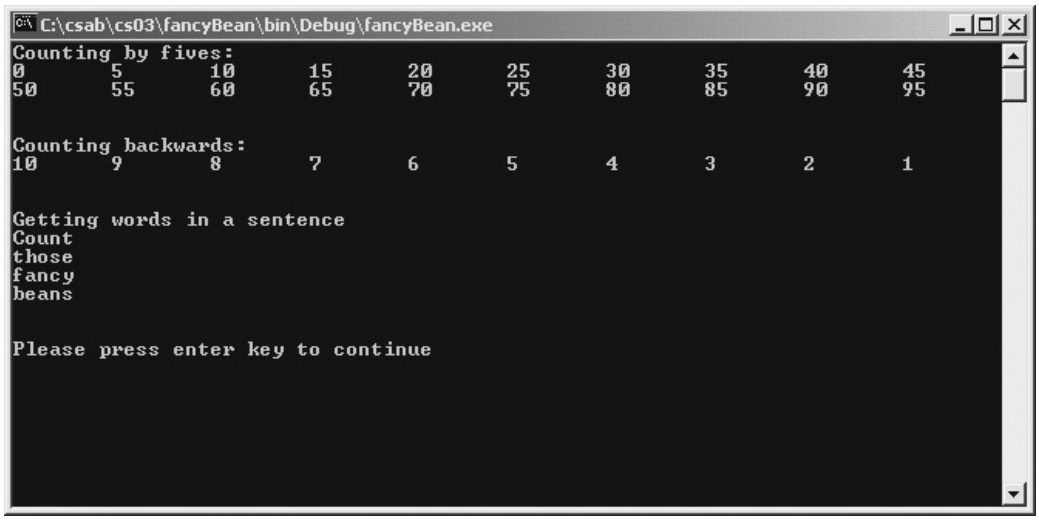

C:\csab\cs03\fancyBean\bin\Debug\fancyBean.exe
Counting by fives:
0 5 10 15 20 25 30 35 40 45
50 55 60 65 70 75 80 85 90 95
Counting backwards:
10 9 8 7 6 5 4 3 2 1
Getting words in a sentence
Count
those
fancy
beans
Please press enter key to continue

Figure 3.6: This bean counter can count by fives, backwards, and pulls the words out of a sentence one at a time.

Take a look at the code for the fancier bean counter, and I'll explain the details:

```
using System;

namespace fancyBean
{
  /// <summary>
  /// Demonstrates a number of variations of the for loop
  /// Andy Harris, 11/29/01
  /// </summary>
  class fancyBean
  {
    static void Main(string[] args)
    {
      int i;
      string sentence = "Count those fancy beans";

      //counting by fives
      Console.WriteLine("Counting by fives:");
      for(i = 0; i <= 100; i += 5){
        Console.Write(i + "\t");
      } // end for loop
      Console.WriteLine();
      Console.WriteLine();

      //count backwards
      Console.WriteLine("Counting backwards:");
      for(i = 10; i > 0; i--){
        Console.Write(i + "\t");
      } // end for loop
      Console.WriteLine();
      Console.WriteLine();

      //demonstrate foreach loop
      Console.WriteLine("Getting words in a sentence");
      foreach (string word in sentence.Split()){
        Console.WriteLine(word);
```

```
      } // end foreach
      Console.WriteLine();
      Console.WriteLine();

      Console.WriteLine("Please press enter key to continue");
      Console.ReadLine();

    } // end main
  } // end class
} // end namespace
```

As you can see, this program has three for loops. Each for loop demonstrates a different variation of the basic for loop.

Skipping Numbers

The first loop in this program counts by 5's to 100 by slightly changing the parts of the for loop statement. In this program, I use the lowercase i as a counting variable.

In the Real World

Although I have told you in other places that 1 is a bad character variable name, there is a grand tradition regarding the use of **i** as a **for** loop sentry variable, especially if that variable will not be used for anything else. The use of **i** in this situation goes all the way back to the earliest versions of FORTRAN, where integer variables had to start with the letters *i, j,* and so on. Even programmers who have never written any significant code in FORTRAN (like me) follow the tradition and frequently use **i** as a generic counting variable. For such a young endeavor, computer programming has a rich history.

The key line for making the program count to 100 by 5's is

```
for(i = 0; i <= 100; i += 5){
```

It means "start i at 0, keep going as long as i is less than or equal to 100, and add 5 to i each time through the loop." The += syntax is similar to the ++ syntax but allows you to add any value to a variable. Therefore,

```
i += 5;
```

is the same as

```
i = i + 5;
```

but the first version is easier to type. Programmers are lazy, so they prefer the more concise syntax.

Counting Backwards

When you understand the basic mechanics of the for loop, it isn't much of a surprise that you can make a loop go backwards. However, you must be careful because a couple things have to change to make a backwards loop. The line that looks like

```
for(i = 10; i > 0; i--){
```

handles the backwards behavior. Notice that I had to start i with a value larger than 0 and that my condition checks whether i is larger than 0. The i–– behavior decrements i, so i–– means the same as i = i – 1.

Using a Foreach Loop to Break Up a Sentence

The for loop has one more variation that is useful in certain circumstances. The foreach loop will be valuable in the Pig Latin program, so I'll show it to you now, even though its full value will be apparent only after you learn arrays in Chapter 8, "Arrays: The Soccer Game." The foreach loop extracts specific elements from a group. The line in the Fancy Beans program that uses the foreach loop looks like this:

```
foreach (string word in sentence.Split()){
```

In a foreach loop, you can use any kind of variable as the sentinel variable, but you must get that variable from a group. The Split() method of a string automatically splits a phrase into a group of words, so this foreach loop repeats one time for each word in a phrase. Each time through the loop, the word variable contains the next word in the sentence. Inside the loop, you can see that I use WriteLine() to print each word on a separate line in the output. When you need to break a phrase or sentence into words, use the foreach loop.

> **Hint** This description of the foreach loop might be unsatisfactory because the foreach loop is dependent on arrays, which you will learn about in Chapter 8, "Arrays: The Soccer Game." You will also see more about the use of the foreach loop in that chapter. In the meantime, simply remember that the foreach code combined with the string.Split() method makes a loop that features each word in a sentence. You will be able to use this feature even if you don't completely understand it.

Using a While Loop

For loops are good for repeating things a certain number of times; however, in some of your programs, though, you won't know how many times something is going to happen. In these instances, you need a while loop. For example, you might need a program to continue asking a question until the user gets the correct answer, or you might write a card program that continues drawing cards until it pulls an ace. In either case, you don't know beforehand how many times the loop will occur, so a while loop is the best solution for this type of situation.

> **Trap** It is possible to use for loops for any kind of looping situation, but using one for indeterminate repetition is a bad idea. For loops are best for situations in which you know how many times something will happen or when you want to take advantage of the counting nature of the loop. For other kinds of looping situations, you use a variation of the while loop.

C# provides another loop, which is driven by a conditional statement (such as the if statement). On the surface, while loops are easier to understand than for loops. However, you need to be careful when you use them because sometimes you can accidentally create a minefield. I'll show you a program that uses a while loop correctly, and then I'll show you what you need to remember to make the loop work correctly.

The Magic Word Program

Here's an illustration that uses a traditional bedtime story to explain loops. In the story, you are seeking a magical treasure. On your way to the magical treasure, you encounter a troll standing in front of a bridge. He will not let you cross the bridge until you utter the magic words. Figure 3.7 demonstrates the troll bridge drama in all its intense glory.

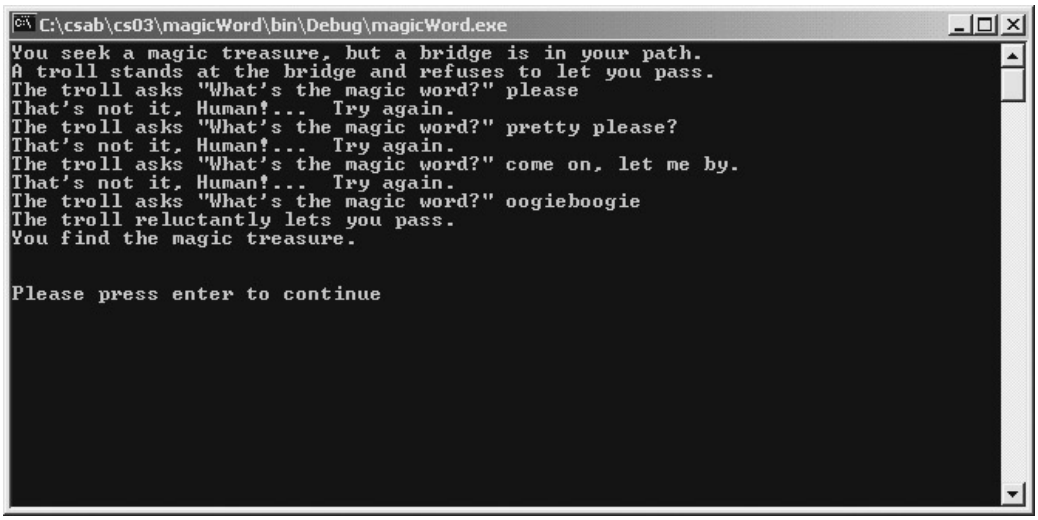

C:\csab\cs03\magicWord\bin\Debug\magicWord.exe
You seek a magic treasure, but a bridge is in your path.
A troll stands at the bridge and refuses to let you pass.
The troll asks "What's the magic word?" please
That's not it, Human!... Try again.
The troll asks "What's the magic word?" pretty please?
That's not it, Human!... Try again.
The troll asks "What's the magic word?" come on, let me by.
That's not it, Human!... Try again.
The troll asks "What's the magic word?" oogieboogie
The troll reluctantly lets you pass.
You find the magic treasure.
Please press enter to continue

Figure 3.7: The troll won't let you pass until you say the magic word.

You can see why a for loop just won't work in this situation. As the programmer, you have no way of knowing how many guesses it will take for the user to figure out the magic word. Here's the source code of the Magic Word program:

```
using System;

namespace magicWord
{
  /// <summary>
  /// Demonstrate while loops by asking for the magic word
  /// until user enters the correct answer
  /// Andy Harris, 11/25/01
  /// </summary>
  class MagicWord
  {
    static void Main(string[] args)
    {
      string theAnswer = "oogieboogie";
      string response = "";

      Console.WriteLine("You seek a magic treasure, but a bridge is in your path.");
      Console.WriteLine("A troll stands at the bridge and refuses to let you pass.");

      while (response != theAnswer){
        Console.Write("The troll asks \"What's the magic word? \" ");
        response = Console.ReadLine();
        if (response != theAnswer){
          Console.WriteLine("That's not it, Human!... Try again.");
        } // end if
      } // end while

      Console.WriteLine("The troll reluctantly lets you pass.");
      Console.WriteLine("You find the magic treasure.");
      Console.WriteLine();
      Console.WriteLine();
```

```
      Console.WriteLine("Please press enter to continue");
      Console.ReadLine();

    } // end main
  } // end class
} // end namespace
```

Most of the program consists of WriteLine commands to give the program some character. The real work happens in the while loop, which looks like this:

```
while (response != theAnswer){
```

A while loop is a very simple construct. The while is followed by a condition. If the condition is evaluated to true, everything in the braces that follows the looping structure is executed. When the code reaches the right brace, it returns to the beginning and checks the condition again. If the condition is true, the code inside the loop happens again. If the condition is evaluated as false, program control passes to the next line after the right brace that follows the loop. Again, use the F11 key to watch the progress of your program so that you can see how the logic flows. Take a careful look at the editor screen in Figure 3.8.

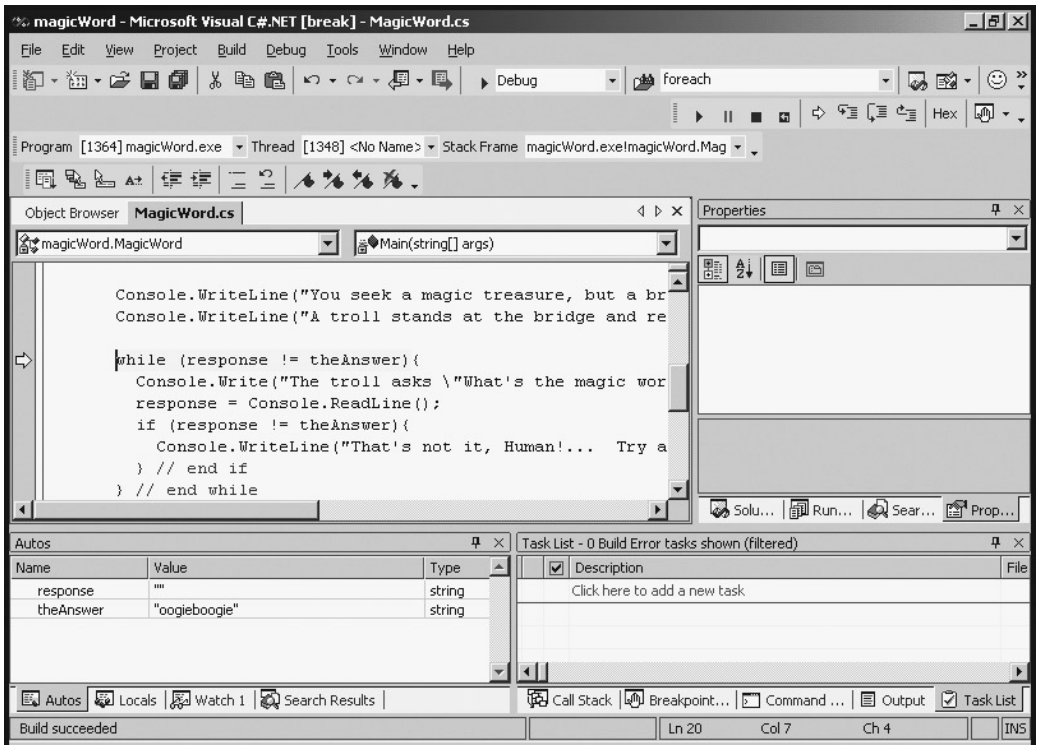


Figure 3.8: The first time through the loop, the condition is false.

In Figure 3.8, you see in the Autos window the values of response and theAnswer. Because these values are different, the condition (response != theAnswer) is true. Therefore, the code inside the loop will execute. Press the F11 key a few more times to see the code continue, and the console window will appear, waiting for a user input. Type in **oogieboogie** (which is the expected response), and press F11 until the while loop line is highlighted again. This situation is pictured in Figure 3.9.

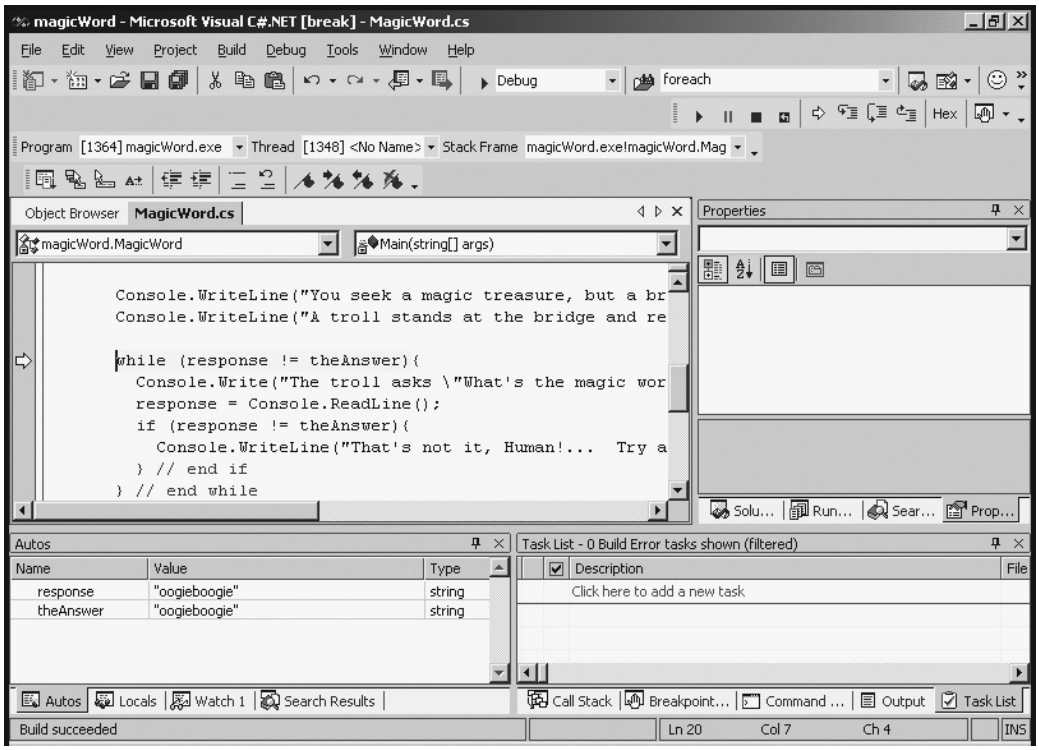


Figure 3.9: Now response is equal to theAnswer, so the condition is false.

As you can see, things look the same, except that the variables response and theAnswer contain the same value. Now the condition (response != theAnswer) is false. Because the condition is false, program control jumps to the next line outside the while loop, which congratulates the user for defeating the troll. Keep pressing the F11 key to verify this behavior until the program ends.

Trick This technique of stepping through the code one line at a time and looking at the values of the variables is called *code tracing*, and it is an invaluable tool. Whenever you are confused (which happens often, even to the pros), use this technique to see what your program is doing. Walking through your programs one line at a time to see what's happening in your variables is one of the best things about tools like the .NET IDE.

Writing an Effective While Loop

While loops are easy to write, but they are also easy to mess up. Following are some of the rules that experienced programmers use to prevent problems with while loops.

Creating and Initializing a Sentry Variable

While loops usually have sentry variables, just like for loops. The sentry variable will be one of the variables in your while condition. Generally, the sentry variable is designed to change while the loop is running. Sometimes you compare the sentry variable to another variable (as I did in the Magic Word game). In that case, you need to think carefully about both variables. Before the while loop starts, make sure that any variables used in the condition have appropriate starting values. If you're not careful, you can have a loop that never runs. Consider this example:

```
string response = "";
while (response != ""){
  Console.WriteLine ("Say something");
  response = Console.ReadLine();
} // end if
```

This loop was intended to keep going until the user typed an empty line (""). However, because the sentry variable begins empty, the while loop skips the first time, and the loop never runs. If your loops *never* run, take a look at the initial value of the variables used in the conditional statement.

Designing an Appropriate Condition

In addition to thinking carefully about the initialization of the sentry variable, consider the condition statement. If the condition is always false, the loop will never run. If the condition is always true, the loop will never quit. You need to devise a condition that will cause the loop to keep going as long as you want and yet leave at the right time. This technique is somewhat of an art form, but there are some steps you can take to make it easier:

- Start by writing the condition in English.
- Convert the condition into a comparison between a variable and a value.
- Define the condition in terms of something that will be true when the loop should execute and false otherwise.

Of course, the condition will use the sentry variable. It may also use another variable, or it may compare against some type of value.

Changing the Variable inside the Loop

Another common problem is to forget changing the value of a variable inside a loop.

Examine the following code fragment:

```
int counter = 0;

while (counter < 10) {
  counter = 1;
} // end while
```

Even though the loop has a good variable and condition, it will *never* exit because the value of counter is set to 1 in every iteration of the loop. Because counter never gets larger than 1, the counter never gets larger than 10, so the loop never exits!

The basic rule is to make sure that inside your loop you have code that makes it possible for the loop to exit. This might seem obvious, but keeping track of this when your programs become large and complex can be difficult. If you encounter an endless loop, use the F11 key to step through the loop, and keep a careful eye on the sentry variable in the Autos window. If you don't have any code inside the loop that allows that variable to change to an appropriate value (one that would make the condition false), your loop will never exit.

In the Real World

Two types of programmers exist in the world: those who have written endless loops and those who will. You will write an endless loop or two in your career, but don't worry. In C#, it's easy to get out of these monsters because you can simply close the console window or click the Stop button in the IDE. However, to see what an endless loop does to your computer, try running the performance meter (press Ctrl+Alt+Delete, and then click the Performance tab) while you have an endless loop running. The simple code in the counting example will use up nearly 100 percent of your processor capacity. Fortunately, it's not quite 100 percent, or your computer wouldn't be able to let you close the program. When your computer bogs down and acts sluggish, often one of the programs

currently running is trapped inside an endless loop.

One way to remember these elements of the while loop is to think back to the for loop syntax. The for loop structure requires you to think about the characteristics of a well formed loop. You are required to define and initialize a sentry variable, define a condition, and ensure that the variable changes to a value that can trigger the end of the loop.

Planning Your Program with the STAIR Process

You are learning a lot of new syntax and many new strategies for thinking about information and programming. However, all these skills and syntax details are not the most important part of the programming process. You can always look up syntax or find a new command on the online help. The most challenging part of the programming process is the creative aspect. You have to be able to visualize a problem and apply a solution to that problem. You also need a framework for your solution so that you don't find yourself staring at a blank screen, wondering what to do next. The next section helps you create that framework.

You don't have to take a class in software engineering or project management to take advantage of good planning strategy for writing your program. I was part of a group of computer scientists at Indiana University which devised a strategy named *STAIR* to simplify the project–planning process. This strategy helps you avoid the headaches of program planning. *STAIR* is an acronym for a general problem–solving technique, but that technique is especially well suited for programming problems. *STAIR* stands for *State the problem, Tool identification, Algorithm, Implementation, Refinement*. I will explain each of these steps now.

S: State the Problem

The first thing you must do is write down the problem in clear, concise English (or whatever your primary language is). This sounds like the easiest step, but it's the most difficult. Your statement of the problem should plainly indicate what you want to accomplish—a sentence or paragraph that describes what the program should do. You should avoid the temptation to get technical here. The actual technologies, commands, and syntax are completely irrelevant at this stage. Your grandmother should be able to read and understand your statement of the problem. (Well, if your grandmother happens to be a hotshot programmer, get somebody else to look at your problem statement!) Later in the process, you will become confused, and it will be helpful to have a record somewhere of what you are trying to accomplish. In C#, I like to include my statement of the problem in the actual code as part of the summary.

T: Tool Identification

After you gain a solid understanding of the problem you want to solve, it's smart to make a list of the tools you will use to solve that problem. Tools can be many things. They can be various commands or objects in the C# language, techniques (such as the random number algorithm I show you in Chapter 2), or structures (such as a while loop). As you gain programming experience, you will learn more tools, and you'll have more practice applying various tools to programming situations. In this book, the program that begins and ends each chapter is devised to illustrate the tools introduced in the chapter. I usually write the names of variables I will need and any major control structures I will use. The exact order and details of how the tools will be used can wait. Often, I list tools that I don't end up using. Golfers usually leave the clubhouse with many clubs in their bags. They can't usually

take every club they own, so they select a set of clubs that are likely to be useful for the course they are playing. Identifying tools is a lot like that.

A: Algorithm

After you identify the possible tools, think about your programming strategy. Programmers call a specific programming strategy an *algorithm*. (Nice sound to it. You have seen by now that programmers love complicated names for simple things.) Now it's time to think about the order in which you will do things. Some programmers use flowcharts or state diagrams to illustrate how the code is supposed to act. For small programs like those in this book, I usually use a simple list of instructions on paper. By the time you get to the algorithm step, you are actually writing down a form of shorthand to record the kinds of steps you will be taking.

Let me emphasize again that the algorithm should be written down. If you work all this out on paper before you turn on the computer, your programming efforts will be much less painful than if you simply sit down and start banging away.

In the Real World

As you begin to program, no doubt you will run into somebody who is an experienced programmer. Very often, that person will say something like "I never plan my algorithms." Although many experienced programmers do not use written documentation for their simpler programs, ask them how long it took them to learn and how many mistakes they made when starting. At professional software development houses, programmers are often not allowed to start working on a program until the plan is developed. You don't have to use STAIR, but if you do not use a scheme to plan your programs before you begin coding, you will run into big problems. If you don't believe me, try writing programs without a plan and see what happens.

I: Implementation

You might be surprised that none of the first three steps require a computer. The best programming happens far from a computer. I wrote the core of one of my most useful programs on the back of an agenda at a very boring meeting. No computer was in sight, but all the real work could be done with paper and pencil. (Don't tell my boss about this—he thought I was furiously taking notes about the meeting.)

The implementation stage begins when you turn on the computer and write code. When most people begin programming, they jump directly to this step. (Instead of the STAIR process, they must use the IR process.) If you can succeed by jumping right into the program without planning, that's fine. Sooner or later, though, you'll be totally clueless about how to proceed. That's when you need to back up and think more carefully about the entire problem. Generally, the implementation phase is the easiest part of programming. If your algorithm is written correctly, translating it into working code is a simple matter. If I look at each line of an algorithm and can say, "I know how to do that," the algorithm is ready for implementation.

Trick Keep your STAIR documentation around. You never know when it will come in handy. I once wrote a program named *ABNIAC* that simulates a simple machine language computer. I got a chance to demonstrate the program on television, but I had written the program in Visual Basic, which works only on the Windows operating system. The television studio used a Macintosh, so my program would

not run. Fortunately, I still had the algorithm on paper. Translating the program from one language to another turned out to be a simple task because the underlying algorithm hadn't changed. (Java runs fine on Macs.) I had already done the hard work of creating an algorithm, so implementing that algorithm again in a new language was all that needed to be done.

R: Refinement

With all this planning, you might expect your program to run correctly the first time. It won't. Even experienced professional programmers expect their programs to crash many times. A lot can go wrong. Keep a positive attitude or else you'll get discouraged. When your program does not work, it usually gives you clues about what is wrong. Sometimes it crashes and gives you an error message. Although this seems bad, it's a good thing because you get lots of information about what has gone wrong. The error messages in C# are reasonably clear, and the editor brings you back to the spot where the interpreter noticed the mistake. These tools can be very helpful in spotting your problems.

In the Real World

The conception of the *STAIR* acronym is a good example of the problem–solving process. I was working with a team of computer science instructors who agreed that we did not have an effective way to teach problem–solving strategies to beginning programmers. We mulled over the principles for days. Although we all agreed on the basic framework, we couldn't figure out how to make it easy to remember and use. The next week, I had a dream about climbing stairs while writing a program, and I suddenly woke up. My wife remembers that I sat up and said, "That's it!!" Then I wrote the acronym *STAIR* on the paper I keep beside my bed. (This wasn't the first time an idea had occurred to me in a dream.)

Although the principles are not at all new, the acronym for thinking about them was an innovation that is now used by many programmers. Perhaps the real lesson is that sometimes your best thinking happens when you aren't trying. Take a break once in a while, and let your mind catch up. Maybe you, too, should keep a pad of paper by your bed.

As a general strategy for refining your program, look back at the STAIR steps again. Did you state the problem correctly? Did you miss a tool that might automatically solve your problem, or did you use a tool incorrectly? Did you define a solid algorithm that will help you solve the problem correctly? Did you implement correctly? Maybe you made a typing error or missed a quotation mark, something silly like that. (In fact, the problems that cause the greatest grief are usually silly, such as a period where a comma should go, misspelled variable names, and the like.)

Applying STAIR to the Pig Latin Program

The STAIR concept is nice in an academic sort of way, but its real power comes when you try to solve a problem of any complexity. To illustrate this, imagine trying to write the Pig Latin program right now, before you finish the chapter. You have all the tools you need, but you don't have a plan. I suspect that it would be a frustrating exercise. Use the STAIR process to set up a plan for your program. This makes the programming experience much more pleasant.

Stating the Problem

The following paragraph explains how I thought about this problem.

Create a program that demonstrates string manipulation and looping structures. The program should ask the user for a phrase and then translate that phrase into pig latin. The program should break the phrase into words and then look at the first character of each word. If the initial character is a vowel, the program should simply add way *to the end of the word. If the first character is a consonant, the program should remove that consonant from the beginning of the word, add the consonant to the end of the word, and add* ay *after the consonant. The program should then ask for another phrase. When the user types* quit *as the phrase, the program should exit.*

Notice how complete and well thought out this statement of the problem is. It could still be improved, but it's a good starting place. Writing a program from a statement like this is much easier than from one that simply states, "Make a Pig Latin program."

Identifying the Tools

This program is likely to need the following tools:

- A while loop to control continuing the program until the user enters *quit*
- Some type of input to get the phrase from the user
- A foreach...split structure to break the phrase into words
- A variable to hold each word (word)
- The String.SubString() method to divide the word into first character and other characters
- Variables for the first character (firstChar) and the rest of the word (restOfWord)
- A condition to see whether the first character is a vowel
- A WriteLine statement to send the completed pig latin phrase to the screen

Other tools might be necessary, but this list will do as a starting point. In essence, I built this list by looking again at the statement of the problem and looking at the kinds of tools I anticipate using to solve the various little problems along the way.

Creating the Algorithm

The algorithm works by restating the problem in terms of the tools. At this point, I usually do two things: I try to get the statements in the correct order, and I begin to flesh out some of the details. This results in a stylistic language that is neither English nor a programming language. It's frequently called *pseudocode* by people who name such things. Although there are some formal definitions for pseudocode, I generally write short statements that I know I'll be able to translate into the language I'm writing in. Here's my first crack at pseudocode for the Pig Latin program: (Note that in this book, pseudocode will be in italics):

Pseudocode:

Create variables, word, firstChar, restOfWord, sentence, pigWord

while user has not said "quit"

    ask user for sentence

    look at sentence one word at a time (foreach)

extract firstChar, restOfWord

if firstChar is a vowel,

pigWord = word + "way"

otherwise

pigWord = restOfWord + firstChar + "ay"

end if

end foreach loop

end while loop

This pseudocode is extremely useful because it provides a blueprint for finishing the program. Implementing the program simply requires fleshing out the pseudocode with C# code. Notice that I still use English–like phrases, such as user has not said "quit". As long as you have an idea how to translate the pseudocode into actual code, using a form of English is fine. However, you often find that you don't know exactly how to proceed on a particular piece of your code. For example, how do you write code to determine whether a certain letter is a vowel? You can do this a number of ways, for example, by using the switch structure and a series of else if clauses. Later, I'll show you another strategy that's more compact. When a part of your algorithm is not ready to implement, you can apply the STAIR process again to that smaller piece of the problem until you are satisfied that you will be able to implement the algorithm.

Implementing and Refining

The implementation and refinement steps aren't really a part of the program–planning process. The implementation is the actual code, which you will see in the next section. Refinement is an ongoing process, so I'll show you how I had to refine the plan as I wrote the actual code.

Writing the Pig Latin Program

The STAIR process led to the development of some pseudocode. Now you have a solid plan, and you know which types of objects, variables, and code structures you will need. It will be relatively simple to translate this plan into a working program.

Setting Up the Variables

With the plan in place, it was easy to figure out a starting point. I did all the normal startup steps and created variables:

```
using System;

namespace PigLatin
{
  /// <summary>
  /// Pig Latin interpreter
  /// Demonstrate loops, string methods
  /// Andy Harris, 11/16/01
```

```
  /// </summary>
  class Pig
  {
    static void Main(string[] args)
    {
      string pigWord = "";
      string sentence = "";
      string firstLetter;
      string restOfWord;
      string vowels = "AEIOUaeiou";
       int letterPos;
```

The program features several useful variables. The pigWord variable holds each word after it is converted to pig latin. The
sentence variable holds the entire original phrase as it comes from the user. The firstLetter variable is used to hold the (surprise!) first letter of each word. The restOfWord variable holds all the other letters in the word. The vowels variable is sneaky. I'll use it to see whether the first letter is a vowel. I had a word variable listed in the algorithm, but it will be created in the section on breaking the sentence into words as part of a foreach structure. The foreach loop requires that you create a variable, so I'll simply create word as part of that structure. Also, I didn't know at first that I needed a letterPos variable, but I added it during refinement. I'll show you how it was used in just a second.

Creating the Outside Loop

The STAIR analysis of this program makes it clear that the program requires two kinds of loops. First, you need a loop to ask for a phrase. That loop will occur every time the user is asked for a new phrase. If the user types *quit*, the program should exit.

```
while (sentence.ToLower() != "quit"){
        Console.WriteLine("Please enter a sentence (or type \"quit\"
to exit)");
        sentence = Console.ReadLine();
```

The main loop is a simple while loop. It checks whether sentence is equal to "quit". As long as the sentence variable is anything but "quit", the program continues. Notice that sentence is initialized to the empty value (" "), which is not quit, so the loop is guaranteed to run at least one time. Also note that I use the ToLower() method to convert whatever the user enters into lowercase. This way, "QUIT", "qUit", and "Quit" will be read as *quit* and cause the program to exit. Add this kind of feature whenever you can.

I then ask the user for a sentence. The sentence variable gets a new value from the screen. Telling the user how to exit is very important. If you don't tell users to type *quit*, they might never guess, and your program would effectively have no end. The sentry variable for this loop is sentence. It is properly initialized, the condition is well conceived, and it has a mechanism for changing the sentry so that the loop can exit. This appears to be a well designed loop.

Dividing the Phrase into Words

The Pig Latin program must act independently on each word in the sentence. You have seen a tool (the foreach loop with the String.Split() method) that performs exactly this function:

```
foreach (string word in sentence.Split())
```

This code sets up another loop, which will repeat one time for each word in the sentence. Each time through the loop, the variable word will contain the value of the current word.

Extracting the First Character

When you are trying to determine tools in an object–oriented language such as C#, start by examining the objects you have on hand. Because the Pig Latin program manipulates strings, it isn't surprising that it uses a number of methods of the string object. I wanted to find a method that lets me extract a substring from a string, and I found it in the String.SubString() method:

```
firstLetter = word.Substring(0,1);
restOfWord = word.Substring(1, word.Length -1);
```

The string manipulation methods came in handy here. The first letter of the word is a substring of the word starting at character 0 and is one character long. Don't forget, computers often begin counting at 0, so the front character of the string is character 0, not 1. For the rest of the word, you need another substring that starts at character 1 and is one less than the total number of words in the string. You can use the Length property to determine how long a string is.

Checking for a Vowel

You can use an if...else if structure to check for a vowel, but this can be tedious, especially if you want to check for uppercase and lowercase values. Instead, I employ the String.indexOf() method as a sneaky way to determine whether the word begins with a vowel:

```
          letterPos = vowels.IndexOf(firstLetter);
          if (letterPos == -1)
{
            //it's a consonant
            pigWord = restOfWord + firstLetter + "ay";
          } else {
            //it's a vowel
            pigWord = word + "way";
          } // end if

          Console.Write("{0} ", pigWord);
```

I check to see where firstLetter occurs within the vowels string. If firstLetter is not in vowels, the IndexOf() method returns a –1, and firstLetter is a consonant. If letterPos is anything but –1, firstLetter is a vowel. I then write out the new word to the console.

Adding Debugging Code

While I was working on the program, I made some mistakes in my logic. I added a few lines that explicitly pointed out the value of various variables so I could clearly see where my logic was flawed. When I had the program working correctly, I added comments to these code lines because if I ever come back to repair this program (say, to deal with blends like *sh* and *th*), I will probably want access to these debugging codes again:

```
//debugging code
//Console.Write("{0}\t", word);
//Console.Write("{0}\t", firstLetter);
//Console.Write("{0}\t", restOfWord);
//Console.Write("{0}\t", letterPos);
//Console.WriteLine("{0}", pigWord);
```

Closing Up the code

This program has a number of structures embedded inside each other. Ensuring that all the code ends properly can be very difficult if you aren't careful about indentation and comments. However, I was careful about these things, so closing up all the code was a straightforward job:

```
        } // end foreach
      } // end while loop
    } // end main
  } // end class
} // end namespace
```

Summary

You learned some very important skills in this chapter. You investigated the for loop, which is used to repeat code a certain number of times. You also looked at how a while loop can be used when the code must repeat an undetermined number of times. You investigated the string object and used some of its methods and properties to do interesting things with text, such as capitalization, searching for a phrase, and determining the length of a string. Finally, you learned how to use the STAIR process to plan a program. In the next chapter, you'll begin to create your own objects.

Challenges

- Write a program that simulates a 10–lap race. Have the program print out the lap time each time through the loop.
- Write a program that asks a user how many dice to roll, rolls a six–sided die that many times, and returns the total and average roll.
- Modify the Math Game from Chapter 2 so that it generates five questions of each type.
- Modify the Math Game so that each question is repeated until the user gets the correct response.
- Create a cartoon simulator that emulates the speech pattern of a cartoon character by replacing all *r*'s with *w*'s so that *You crazy rabbit* becomes *You cwazy wabbit*.

Chapter 4: Objects and Encapsulation: The Critter Program

You have learned most of the critical aspects of programming a computer. From now on, you will be learning variations of these basic skills. To help you manage the complexity of larger programs, C# uses an important programming paradigm called *object–oriented programming* (OOP). You are already slightly familiar with object orientation because you've been using objects that come with the .NET environment, such as the string object, convert object, and console object. However, you will discover the real power of OOP when you create objects of your own. In a nutshell, object–oriented programming enables you to combine the information and instructions of your programs into *objects*. The objects you have already used feature properties, which encapsulate the data in a program, and methods, which house the instructions in a program. You will learn how to build your own objects with these properties and methods. After reading this chapter you will be able to

- Create your own objects.
- Add methods to your program.
- Communicate to and from your methods.
- Build a classic console–based menu system.
- Trap for certain kinds of errors.
- Create your own custom object.
- Add methods and properties to your object.
- Understand how basic scope modifiers work.
- Use properties to improve the reliability of your object.

Introducing the Critter Program

Like most of the examples in this book, the main program for this chapter has little practical merit but is entertaining. You'll build a virtual pet named a *critter.* The critter is very simple, but as you see in Figures 4.1 through 4.5, it's fun to interact with.

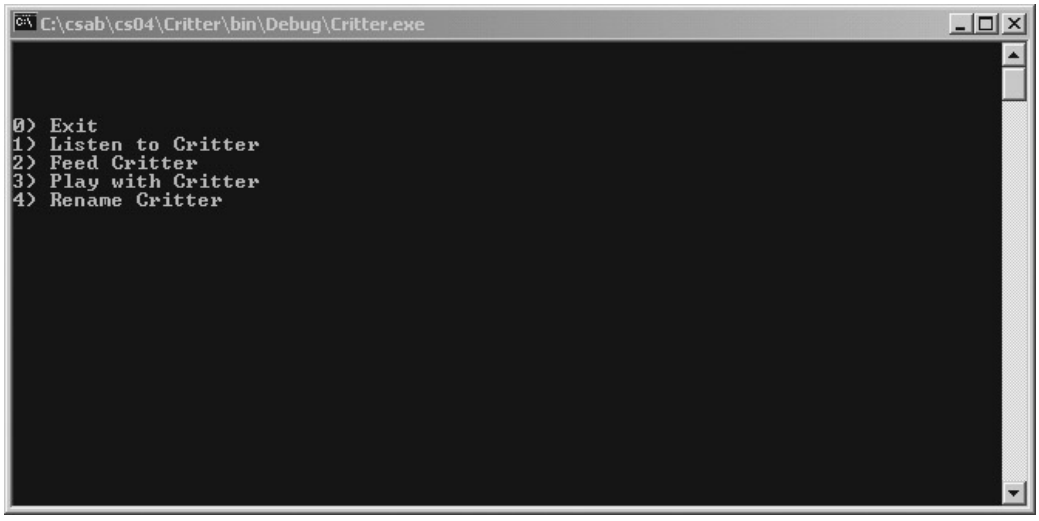


Figure 4.1: Introducing your critter. Go ahead and talk to it!

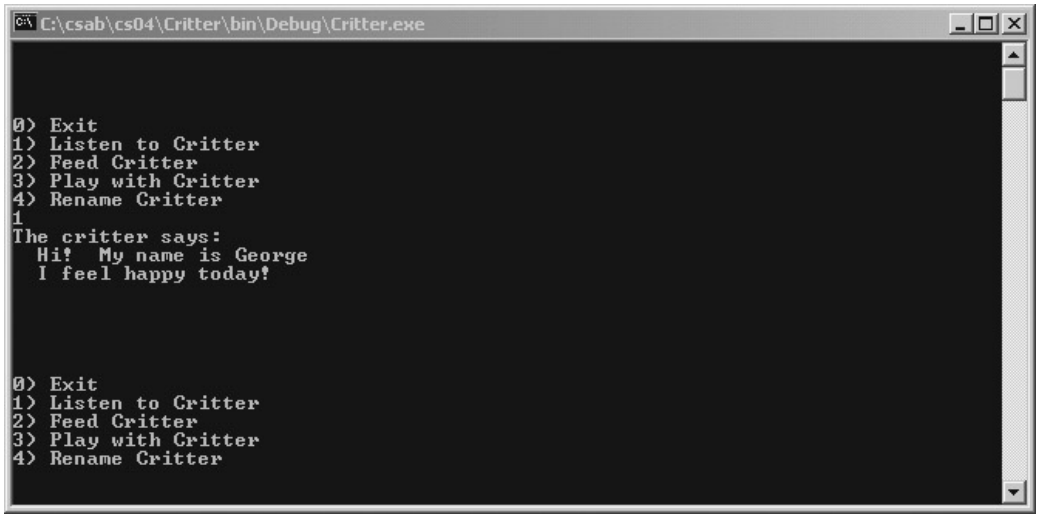


Figure 4.2: The critter tells you about itself.

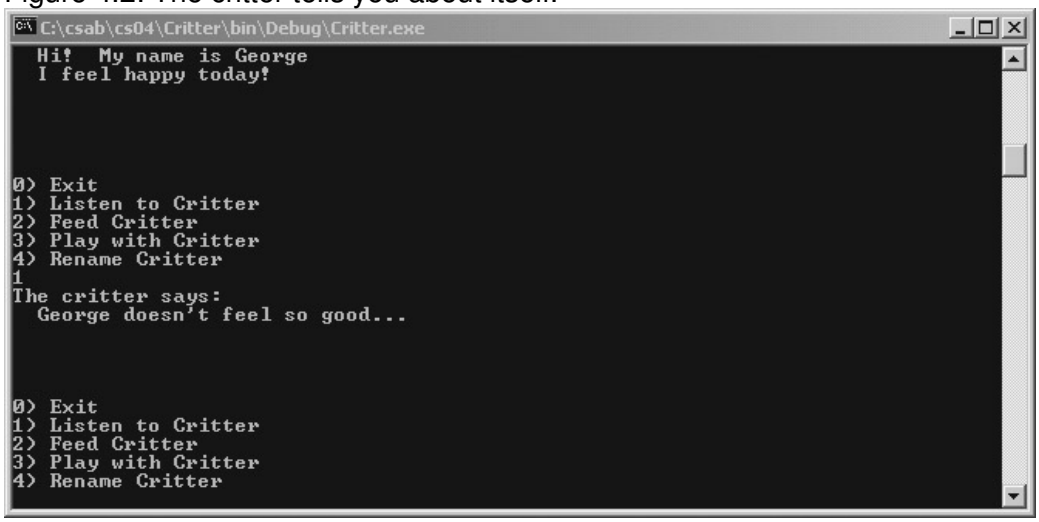


Figure 4.3: After talking for a while, the critter becomes melancholy.

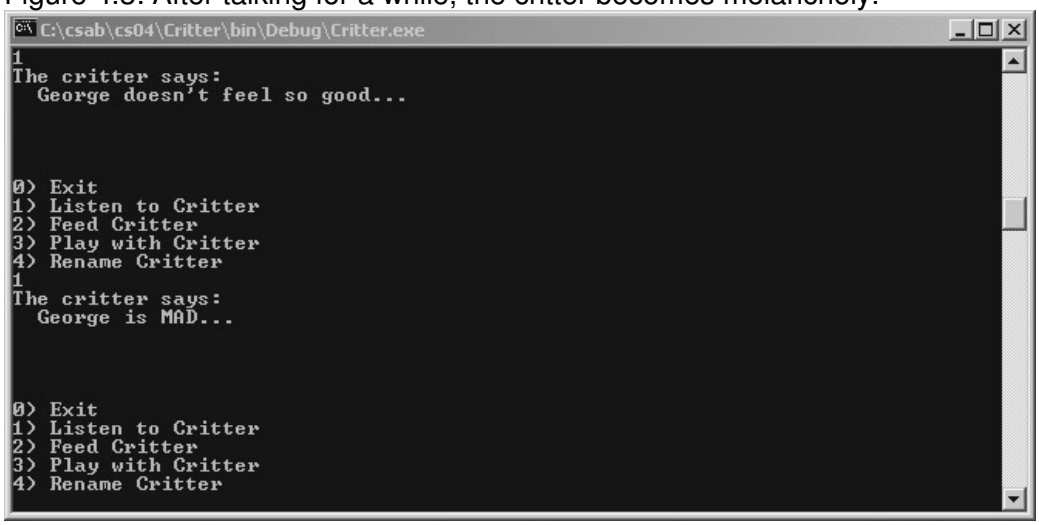


Figure 4.4: Without appropriate attention, the critter becomes angry.

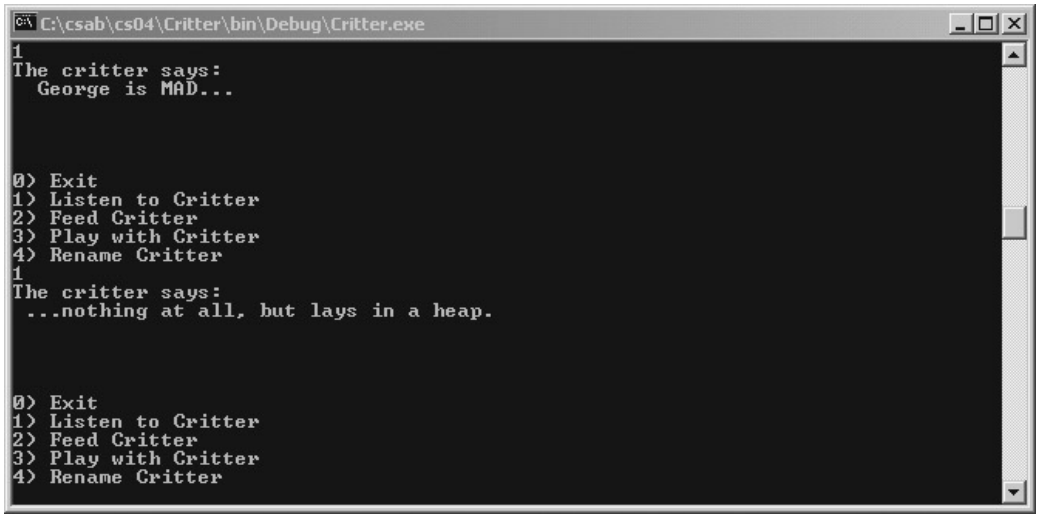


Figure 4.5: The cost of neglect can be a sad and lifeless critter. Fortunately, with enough food and love, it can be revived.

I used simple rules to drive the critter's behavior. During each turn, the critter ages and becomes hungrier and a little more unhappy. Whenever you talk to the critter, it gives you a message based on its current happiness level. If the critter is hungry, it becomes unhappy even more quickly. Playing with and feeding the critter make it happier. That's all there is to the design of the critter, but to create the program, I relied on OOP techniques. The critter is actually an object, and it can talk, eat, play, and age.

Another interesting feature of the Critter program is the use of a simple menu. Many console–based programs use a menu, so now you'll learn how to build one.

Creating Methods to Reuse Code

As programming has evolved, computer scientists have discovered techniques for making it easier to write and maintain programs. Most of these ideas are borrowed from the outside world and are simple. One of the most important is the notion of *encapsulation*, another word for *grouping together* (as already mentioned, computer scientists like to give complicated names to simple ideas).

For example, if somebody asked you what you did this weekend, you might reply, "I played with the kids, went shopping, and worked on the house." You probably would not describe *everything* you did over the weekend. Instead, you would group several complex activities into one phrase. You would say, "I played with the kids," without describing all the components of this behavior (which, in my house, involve loud noises, wrestling, Dad's dressing up in at least one ridiculous outfit, and somebody in tears). All those details are wrapped up in one phrase.

Encapsulation in programming works much the same way. You take groups of instructions and put them together to make methods. These methods are things the object can do. For example, the critter can eat and play. It has Eat and Play methods to enable these behaviors. After you've defined a set of instructions as a method, you can refer to the name of the method, and all the code related to that name will execute.

The Song Program

To clarify the concept of encapsulation, I wrote a program that replicates a serious and weighty application of your computer's horsepower. The Song program repeats the words to the classic

children's song *This Old Man.* In case it has been a while, the words (for the first couple of verses) go like this:

This old man, he played one
He played knick–knack on my thumb
With a knick–knack paddy–whack
Give a dog a bone
This old man came rolling home

This old man, he played two
He played knick–knack on my shoe
With a knick–knack paddy–whack
Give a dog a bone
This old man came rolling home

The song goes along for 10 verses, but the pattern is evident by the second stanza. You can see that most of the verses are almost the same, except that each features a number and the old man plays knick–knack (whatever that means) on something that rhymes with the number. A computerized version of the song is amazingly compact if you think of it in terms of methods.

Building the Main() Method

So far, all the programs in this book have been written entirely in the Main() method. You might be surprised to learn that the Main() method of most programs in C# is very small. Figure 4.6 demonstrates the Main() method for the Song program.

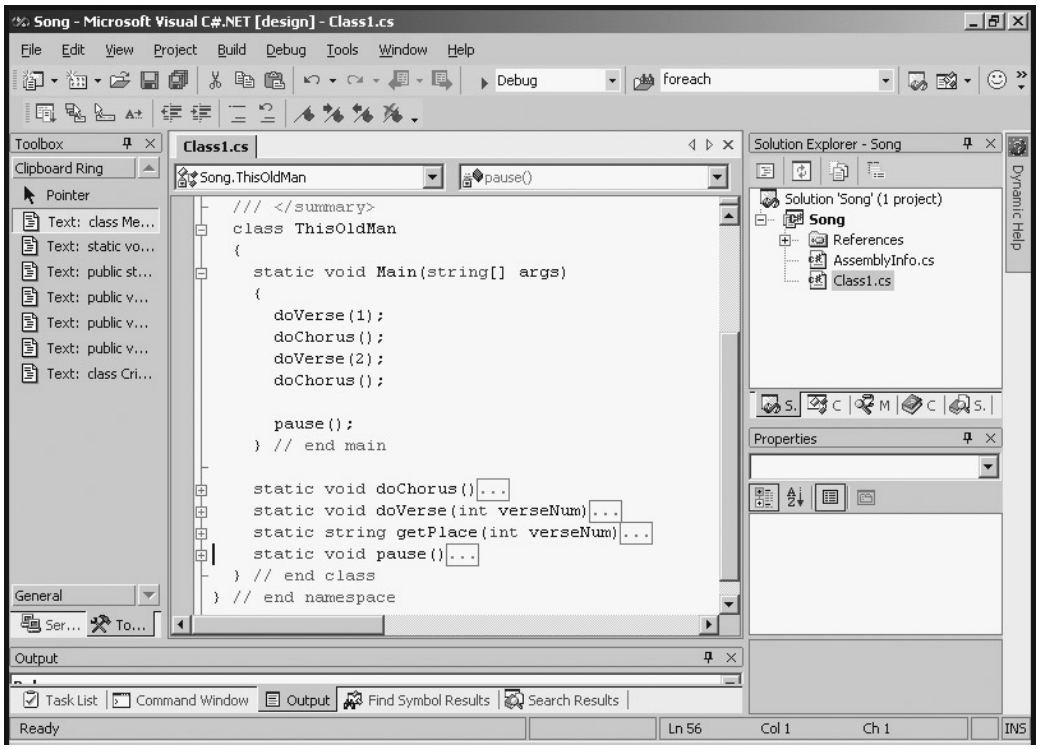


Figure 4.6: The Main() method features a few commands. You can probably guess what each one does.

Like most songs, *This Old Man* has a pattern: a simple verse that changes and a chorus that stays the same. You can summarize the pattern of the song like this:

verse 1

chorus

verse 2

chorus

Of course, this summary is much like the main() method. To pull this off, I had to build methods to handle the verses and the chorus, but all the code is delegated to these methods. With the details encapsulated away in various methods, the main() method clearly demonstrates the main flow of the program.

Trick In Figure 4.6, I utilized a feature of the .NET IDE. For this image, I wasn't interested in the details of any methods except the main() method, so I used the small plus and minus signs along the left margin to collapse and expand the methods I wanted to look at. This is a wonderful tool because it enables you to focus on the parts of the program you're currently working on, without losing sight of how each part fits into the larger picture.

Of course, the program won't work exactly like this yet because it is necessary to write the methods. You do that in the next few sections.

Creating a Simple Method

You will start with the doChorus() method because it's the most straightforward:

```
static void doChorus(){
  string message = "";
  message += "\n...With a knick-knack paddy whack\n";
  message += "Give a dog a bone\n";
  message += "This old man came rolling home.";
  message += "\n\n";
  System.Console.WriteLine(message);
} // end doChorus
```

Custom methods can begin just like the main() method. I declared the method with

```
static void doChorus(){
```

The static keyword defines the method as one that can be called before the class has been completely created. It's okay if you don't understand that yet. The concept will make more sense after you learn about constructors and instantiation later in this chapter.

The void keyword indicates that this method will not return any value. Again, you have to take my word for it, but I'll explain exactly what that means later in this chapter in a section cleverly called "Returning a Value." doChorus() is the name of the method, so all the code in this section is named doChorus, and the main() method can invoke all these commands by simply calling the doChorus() method.

The code inside the method is straightforward. I created a string variable named message, added some values to it to write the chorus, and wrote that message to the screen.

Notice that you can create variables inside a method, as well as inside a class. This is important because a variable lives only as long as the structure in which it's created. In other words, the

message variable is created each time the doChorus() method is invoked, but as soon as that method finishes (which takes a fraction of a second), the message variable is destroyed. This is good because you don't have to worry about the message variable's interference with other code you're writing. This local ownership of variables is called *data hiding*, and it's another benefit of encapsulation.

In the Real World

A long time ago I wrote a program for a middle school. The program had to be run on Apple IIe machines, so I was required to use an antiquated language that allows only two–character variable names and cannot encapsulate code. I had the program running almost perfectly, but it was doing very strange things at unpredictable intervals. After tearing out my hair for three days, I finally discovered that I had accidentally used the same variable name for two different things. This is exactly the sort of situation that encapsulation and data hiding can prevent.

Adding a Parameter

The doChorus() method is easy to write because it works exactly the same way each time you call it. However, you frequently run into situations more like the verse—you want the program to behave almost the same each time, but you want to send it a value to act upon. Many of the methods you've already used (such as Console.WriteLine) work in this way. They expect you to send a value between the parentheses, and then they act on that value. Notice in the main() method that, when I call the doVerse() method, I always include an integer value. It is quite easy to write a method that accepts a value:

```
static void doVerse(int verseNum){
  string message = "";
  message += "This old man, he played ";
  message += verseNum;
  message += ". \nHe played knick-knack ";
  message += getPlace(verseNum);
  Console.WriteLine(message);
} // end verse
```

To modify a method so that it can accept a value, put a variable declaration inside the quotes that follow the method's name. This special variable is called a *parameter*, and it automatically accepts whatever value was sent when the method was called. For example, when the main() method says doVerse(1), the value of verseNum inside doVerse() is 1. The next time the main() method invokes doVerse(2), the verseNum parameter will have the value 2. You can have as many parameters as you like, but you must indicate the type of parameter you will send, and you must separate the parameters by commas. Parameters have the same kind of limited life span as variables declared inside a method.

Notice that the doVerse() method also has a message variable, but this one is entirely unrelated to the message variable in doChorus().

In the following line, you can see how I used verseNum to integrate the verse number into the verse:

```
message += verseNum;
```

However, associating the place with each number (such as the thumb for verse 1 and the shoe for verse 2) is more complex, so I decided to ship that code to another method. Take a look at this line, and you'll see that it's a little different:

```
message += getPlace(verseNum);
```

This sends the value of verseNum to another method, named getPlace(), but the getPlace() method seems to work differently than doVerse() and doChorus(). It doesn't stand on its own in a line of code, but it returns a value you can do something with. In this case, the results of the call to getPlace(verseNum) will be appended to the message variable.

Returning a Value

You can write methods that receive values through parameters and methods that send values through a return statement. The Console.ReadLine() method is probably the classic example of the latter kind of method. You can usually spot methods that return a value by the way they are used. The new value is usually printed to the screen or assigned to a variable. Methods that don't return a value are usually used in a line of code by themselves, such as doChorus(). I designed the getNumber() method so that it will accept a parameter (the verse number) and return the associated place. Here's how it works:

```
string message = "";
  switch (verseNum){
    case 1:
      message = "on my thumb ";
      break;
    case 2:
      message = "on my shoe ";
      break;
    default:
      message = "not yet defined";
      break;
  } // end switch
  return message;
} // end getPlace
```

The code works as a simple switch statement. The message variable (again, unrelated to the other message variables because it's defined locally in a method) gets some value based on the verse number. The value of message is returned in the last line of the method. Any code that invokes this method will receive the value of message.

Trick Use the F11 key trick to watch the flow of code through this program. It's very important to see how the logic flows between methods. Mastering this trick is the key to building more complex programs successfully.

Figure 4.7 illustrates the output of the song as produced by the Song program.

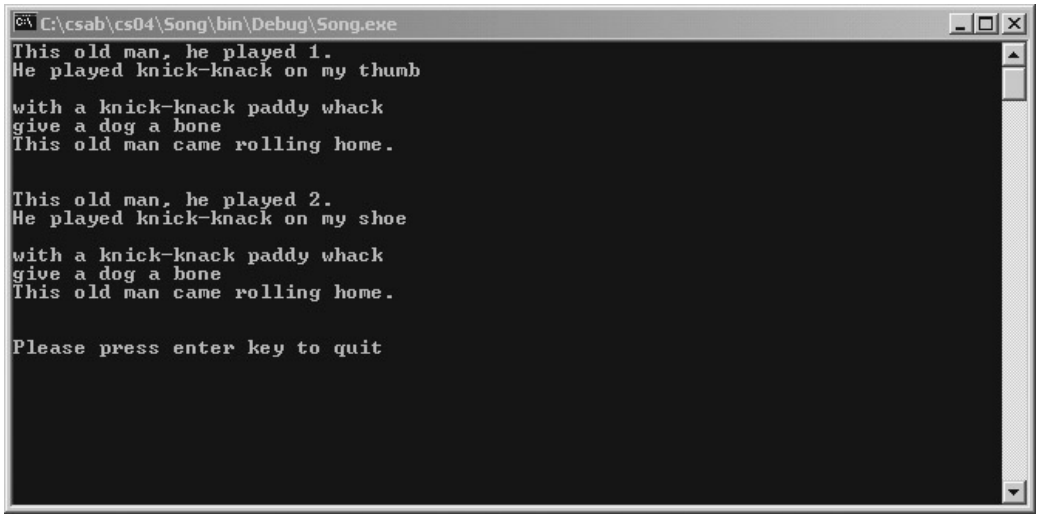

C:\csab\cs04\Song\bin\Debug\Song.exe
This old man, he played 1.
He played knick-knack on my thumb
with a knick-knack paddy whack
give a dog a bone
This old man came rolling home.
This old man, he played 2.
He played knick-knack on my shoe
with a knick-knack paddy whack
give a dog a bone
This old man came rolling home.
Please press enter key to quit

Figure 4.7: The Song program re–creates the song *This Old Man*.

Creating a Menu

Menus are a mainstay of console–based programming and a great place to practice your method–building skills. You can begin to build the Critter program by building the menu structure for it. In Figure 4.8, you can see the output of the Critter Menu program described in the next section.

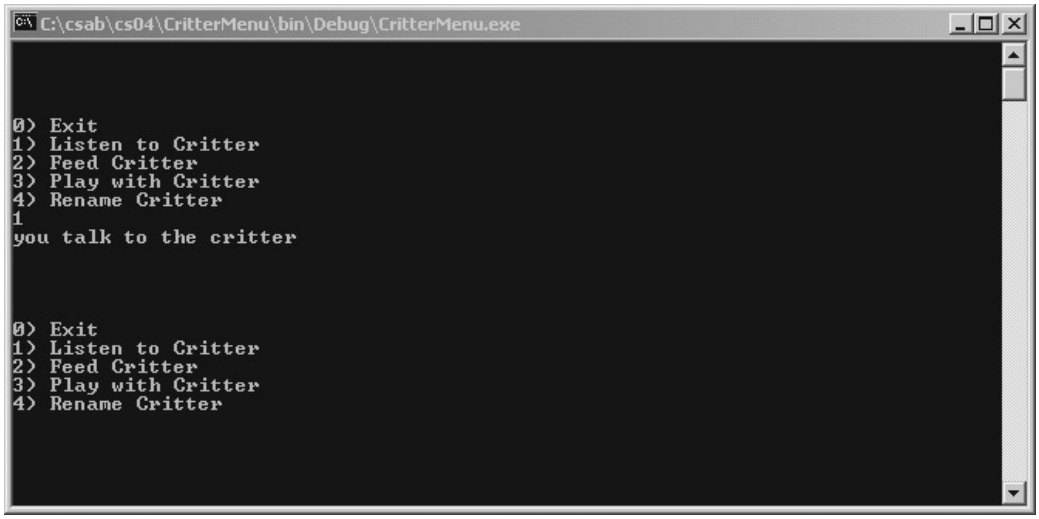


Figure 4.8: The menu for the Critter program is a standard console menu.

Creating a Main Loop

When programmers use a method, they often refer to it as *calling* the method. The Main loop calls a method to display the menu and then uses a switch statement to perform various tasks, based on the results. The entire assembly is enclosed inside a while loop, so the program repeats until the user exits. This is a very common scheme for this type of program. Here's the code for the Main() method of the Critter Menu program:

```
using System;

namespace CritterMenu {
  /// <summary>
  /// Critter Menu
  /// Build a basic menu structure
  /// Andy Harris, 12/13/01
  /// </summary>
```

```
class Menu {
  static void Main(string[] args) {
    bool keepGoing = true;
    int choice;

    while (keepGoing){
      choice = showMenu();
      switch (choice){
        case 0:
          keepGoing = false;
          break;
        case 1:
          Console.WriteLine("you talk to the critter");
          break;
        case 2:
          Console.WriteLine ("You have fed the critter");
          break;
        case 3:
          Console.WriteLine("You have played with the critter");
          break;
        case 4:
          Console.WriteLine("You have renamed the critter");
          break;
        default:
          Console.WriteLine("That was not a valid input");
          break;
      } // end switch
    } // end while loop
  } // end main
```

Creating the Sentry Variable

I created a boolean variable named keepGoing, which is initialized to true. The program continues to cycle until the value of keepGoing is evaluated to false. This technique has a couple advantages. First, figuring out what's going on is easy because the variable name is closely related to the concept. Setting keepGoing to false causes the program to stop. Also, several other situations can cause the program to end. Rather than keep track of all these in the Main loop, I like to use one sentry variable. Then, any time I want to end the program, I can simply set keepGoing to false, and the next time the loop executes, the program will end.

Calling a Method

The main() method doesn't display the menu. It sends this work to the showMenu() method, which is designed to return an integer indicating which menu item the user chose. I will describe the showMenu() function in a moment. For now, in the spirit of encapsulation, assume that it works correctly and follow the logic of the Main() method.

> **Trick** The ability to ignore the details of a subprogram temporarily is a useful skill and very much in keeping with the ideals of encapsulation. The concept is to focus on one problem at a time. At first, you think about how the main menu logic will work. After that is functioning correctly, you think about how it will relate to the showMenu() function. You can write complex programs only by dividing them into more digestible pieces, which is one of the tasks encapsulation does for you.

Working with the Results

After you receive a value from the showMenu() method, you need to do something with that value. I used a switch statement to determine which course of action to take. Also, notice that I included a default clause—because users do crazy things. You have to anticipate that they will type values you aren't asking for, and you must be able to respond to those situations.

> **Trick** Sometimes you set up a situation and think that you know every possible outcome of a calculation or user request. Then you are tempted to skip the default clause. Don't skip it! I've been surprised many times to find that something I thought was impossible occurred (usually because I didn't anticipate a particular set of circumstances). Having a default clause that is never called beats not having it and watching your program crash when something unexpected occurs.

Writing the showMenu() Method

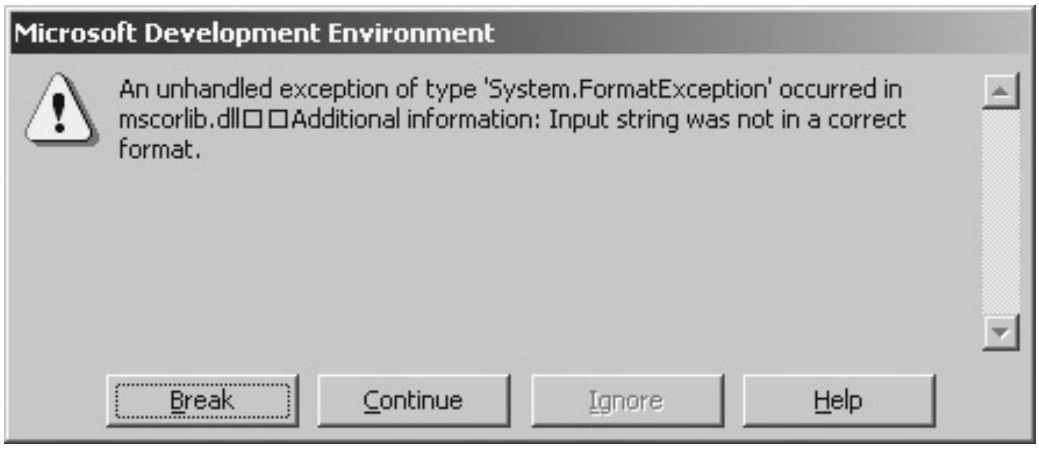
The remainder of the Critter Menu program is dedicated to displaying the menu. This is a very simple method:

```
    static int showMenu(){
      int input = 1;
      Console.WriteLine();
      Console.WriteLine();
      Console.WriteLine();
      Console.WriteLine();
      Console.WriteLine("0) Exit");
      Console.WriteLine("1) Listen to Critter");
      Console.WriteLine("2) Feed Critter");
      Console.WriteLine("3) Play with Critter");
      Console.WriteLine("4) Rename Critter");
      try {
        input = Convert.ToInt32(Console.ReadLine());
      } catch (FormatException) {
        Console.WriteLine("Incorrect input");
        input = 1;
      } // end try
      return input;
    } // end showMenu
  } // end class
} // end namespace
```

I wrote the menu to the screen and read a value from the console. However, when I tested my program, I found it easy to crash.

Getting Input from the User

The input variable is an int, and Console.ReadLine() returns a string, so I had to convert the string to an integer with Convert.ToInt32(). (If you don't remember how this object works, review the discussion of the convert object in Chapter 2, "Branching and Operators: The Math Game.") When I ran the program a few times, I found that it worked well as long as I typed a number. If I typed a letter, the program crashed and gave an unfriendly error message to the user, as you can see in Figure 4.9.


Microsoft Development Environment
An unhandled exception of type 'System.FormatException' occurred in mscorlib.dll□□Additional information: Input string was not in a correct format.
Break
Continue
Ignore
Help
Figure 4.9: This error message makes my program seem very unfriendly.

Trap You must test your programs thoroughly. Better yet, have typical users test them. You might have heard this saying: *The hardest part of making things foolproof is that fools can be so ingenious.* Although your users aren't fools, they probably aren't programmers either. They try things you might never think of, such as typing a *k* (or more likely, the entire line of a menu or something like "I don't know. You choose for me"). You must anticipate these problems. Fortunately for programmers, the exception–handling features described in the next section make trapping for errors much easier than it used to be.

Handling Exceptions

Making your programs as friendly as possible to the user is important, so you must avoid situations that cause the program to crash. Fortunately, C# has a robust set of features, called *exceptions,* that will help you. Exceptions occur when something happens that the code doesn't know how to handle or process. For example, if the user types a *k* and the convert object doesn't know how to convert a *k* to an integer, it throws an exception. This is like the warning indicators on your car's dashboard. In most cases, when your program hits an exception, it will stop running and tell the user what kind of exception occurred. Of course, the user doesn't care about this. He or she simply wants the program to work correctly.

Fortunately, there's an easy solution. When you are testing a program and find an exception, you can surround the part of your code that caused the error with a try { } catch { } block, as in the following code fragment:

```
try {
  input = Convert.ToInt32(Console.ReadLine());
} catch (FormatException) {
  Console.WriteLine("Incorrect input");
  input = 1;
} // end try
```

The following pseudocode is what the code means:

```
Try to convert the console input to an integer
If you encounter a FormatException,
  Tell the user his input was incorrect
  Reset the input to 1, which is definitely a legal value
```

Any time you encounter an exception when testing your code, consider adding an exception–handling block. The try { } clause surrounds the code you suspect will cause trouble. As in other situations that use braces, you can have as many lines of code as you want inside the

braces. However, you must isolate any code you think will cause problems so that you don't end up trapping for the wrong error. The catch statement describes which type of exception you are expecting to encounter. The code in braces following the catch statement is the special code you run if an exception occurs. This typically involves informing the user that something went wrong and taking action to rectify the situation.

Returning a Value

The entire point of the showMenu() method is to get a numeric value from the user and return this value to the user. By the last line of the method, you are guaranteed to have an integer value in the input variable, so you return that value to whatever program called the showMenu() method.

Creating a New Object with the CritterName Program

You have begun to understand the benefits of OOP, but the real fun begins when you create your own objects. Although making the Critter program work without using OOP principles is possible, the point of this chapter is to show how to create objects, so that's what you do next.

Figure 4.10 shows the output of the Critter Name program you create in this section.

Figure 4.10: This version of the Critter program features a critter that knows its name.

Creating the Basic Critter

A critter is an object. Objects, as you recall, can have properties, methods, and events. An object has a *class* (which is like a recipe—it defines the cookie but isn't a cookie) and *instances* (the cookies made with the recipe). To create a critter object, you have to create a class and make an instance of that class. You can start with a very simple version of the Critter class. Later in the chapter you'll give it more capabilities, such as changing its name and talking. Here's the code for the simpler Critter class:

```
using System;

namespace CritterName {
  /// <summary>
  /// Critter Name
  /// Creating a simple class
  /// Andy Harris, 12/13/01
```

```
  /// </summary>

  class Menu {
    static void Main(string[] args) {
      ...
    } // end main

    static int showMenu(){
      ...
    } // end showMenu
  } // end class

  class Critter {
    public string name;
  } // end class

} // end namespace
```

Note that I collapsed the Main() and showMenu() methods in this code listing—because there are very few changes in the code (I'll show you those changes shortly) and because I wanted you to see the general structure of the program. For the first time, your program has more than one class in it. You can add as many classes to a project as you want. The Menu class is largely the same, but the Critter class is new. You create a new class with the class keyword, followed by the name of the class and a pair of braces ({ }). Classes do not have parentheses because they do not accept parameters.

The properties, methods, and events a class owns are collectively known as *members*. This version of the Critter class has only one member, name.

Using Scope Modifiers

A scope modifier is a special word used to determine whether a method or variable is visible to other methods and classes.

When you look back at the code for the Critter class above, the definition of the variable name is unique. It looks like most variable definitions, except that it starts with the keyword public. This term indicates that the variable name should be available to elements both inside and outside the Critter class. As you'll see momentarily, this is not such a great idea, but I wanted to show you the simplest kind of class first.

The private keyword is an example of a scope modifier. C# supports a number of scope modifiers, but for now, you need to know only two. If you want a variable to be available outside the class, you declare it as public, as I did with name in this example. If you want the variable to exist only inside the class, you either declare the variable private or leave out a scope modifier (because the default scope is private).

Using a Public Instance Variable

Because the name variable is declared inside the Critter class and is used by instances of that class, it's an example of an *instance variable*. Instance variables are variables defined inside a class but outside any method definitions. All this terminology seems overwhelming, but you'll run across it later, so you had better learn the lingo. In any case, the formal description of name would be "a public instance variable of the Critter class." Right now, all the critter object has is this one public instance variable. This makes the class very simple to build, but it has some limitations. You'll improve the Critter class throughout the chapter, but at least it works for now, so you can

concentrate on how to use a custom class inside your programs.

Creating an Instance of the Critter

The Main() method of the CritterName program is almost identical to that of the Critter Menu program, except that I added two lines before the while loop:

```
static void Main(string[] args) {
  bool keepGoing = true;
  int choice;

  Critter myCritter = new Critter();
  myCritter.name = "George";

  while (keepGoing){
    ...
  } // end while loop
} // end main
```

Again, I removed some of the code (specifically, the while loop) so that you can concentrate on the part of the program that is new. The complete code is available on the CD−ROM accompanying this book.

The line containing new Critter() creates a new instance of the critter object and assigns it to a variable named myCritter. After you define a class, it is essentially a new variable type, and you can create a variable of that type. The new keyword informs the computer that you will be making an instance of the Critter() class. Critter is the class (or cookie recipe), and myCritter is an instance (a cookie) of that class. In most situations, you work not with the class but with instances of the class. Generally, whenever you create a custom class, you also make an instance (or several instances) of that class. In the next line, you'll see that you can assign a value to myCritter.name. In fact, when you type myCritter into the IDE, a pop−up menu appears, listing all the possible members of myCritter. You can choose name from this automatic menu in the same way you choose WriteLine or ReadLine from the console object's automatic menu system.

Referring to the Critter's Members

I also made a modification to the switch because I wanted the critter to say its name when you talk to it. Take a look at the appropriate section of the switch statement in the Main() method:

```
case 1:
  Console.WriteLine("The critter says: my name is {0}", myCritter.name);
  break;
```

This code refers to the name member of the critter object. After you define an object and add members to it, using the object is easy. You don't have to know exactly how the member is defined or works. You simply refer to it. This is another example of the joys of encapsulation.

Adding a Method

Objects are more interesting when they can do something. As you recall, *methods* are actions that objects can perform. Now you'll add a method to the critter so that it can do something interesting. Adding a method to a class is just like adding methods in ordinary programs.

Creating the talk() Method for the CritterTalk Program

Take a look at the next version of the Critter program, Critter Talk. This version is on the CD−ROM as *CritterTalk*, if you want to look at the entire program.

```
class Critter {
  public string name;

  public void talk(){
    Console.WriteLine("The Critter says: My name is {0}", name);
  } // end talk

} // end class
```

I added a method named talk() to the Critter class. The method is very simple. It sends a message to the screen. Note that, inside the method, the program refers to the name variable. It can do this because both the name variable and talk() method belong to the Critter class.

Changing the Menu to Use the talk() Method

Because the Critter class now has a talk() method, the main program can invoke myCritter.talk() to hear from the critter. Examine how I changed the code in the main() method to take advantage of the critter's newfound verbal skills:

```
case 1:
  myCritter.talk();
  break;
```

I didn't change anything else in the program. You might notice that the main program becomes a little simpler as some of the functionality is shuffled off to the object. In object−oriented programming, as much detail as possible is handled by objects rather than by the main program.

Creating a Property in the CritterProp Program

The public instance variable already created for name was easy to make, but it has some serious weaknesses. Generally, you want to write your programs to prevent problems. By making the name public, you have no way to ensure that it will get an appropriate value. Figure 4.11 shows the next version of the Critter program, Critter Prop, featuring a characteristic called the *property*.

Figure 4.11: You can change the name property so that the critter will reinforce special rules.

Also, you might want a characteristic to be read–only so that it can be changed only from within the class. For example, if you have a circle class with a radius property, users shouldn't be able to set the area of the circle. They might set both values to 1, which should not be done. A circle with a radius of 1 must have an area of 2 * pi. It would be better for the area property to be read–only so that the user cannot change it inappropriately. Having many public variables hanging around is also considered dangerous because they are vulnerable to being changed inadvertently.

Instead of public instance variables, you can use a special entity of objects: the property. You've seen properties before, for example, the length property of a string object, which is written myString.Length. *Properties* are special characteristics of an object. They appear to be variables directly attached to a particular type of object. Critter.name seems like a property because you can refer to it like a property. However, you will see shortly that there is a better way to describe properties than simply assigning a variable to a class. Properties enable you to control access to information inside your class, which is a very powerful capability.

Examining the Critter Prop Program

I made one simple change to the menu code. Because I know that I'll be changing the critter's name, I implemented the code for changing the name in the menu:

```
case 4:
  Console.WriteLine("Current name: {0}", myCritter.name);
  Console.Write("Change name to: ");
  myCritter.name = Console.ReadLine();
  break;
```

Notice that this code is exactly the same, whether the Critter class uses a private instance variable or a property. One of the hallmarks of good object–oriented programming is that you improve an object's performance (as you will by adding a property) without requiring numerous changes to the programs that use the object.

Creating the Critter with a Name Property

I have made a number of changes to the Critter class, all of which involve the critter's name. The name of the critter is stored in a private variable, which is accessed through some special methods. Take a look at the new code for the Critter class:

```
class Critter {
```

```
  private string pName;

  public string Name {
    get {
      return pName;
    } // end get
    set{
      if (value.Length > 8){
        Console.WriteLine("The name can't be more than 8 characters");
        pName = value;
        pName = pName.Substring(0,8);
        Console.WriteLine("Changing name to {0}", pName);
      } else {
        pName = value;
      } // end if
    } // end set
  } // end string property

  public void talk(){
    Console.WriteLine("The Critter says: My name is {0}", Name);
  } // end talk

} // end class
```

Although the changes seem extensive, they make a lot of sense. To make a property, you create a private instance variable, and then you add special methods to your class to provide access to the private variable. I'll describe these changes next.

Making a Private Instance Variable

The first thing I did was eliminate name as a public instance variable. (Remember, it's called an *instance variable* because its value has meaning to the entire instance of the class.) It's public because another program or programmer can have access to its value. I replaced the public instance variable Name with a private instance variable named pName. This private variable can only be accessed from within the Critter class.

Hint I used the p in pName to remind myself that this is a private instance variable. Many programmers use this convention so that they don't confuse the public property Name with the private instance variable pName. Microsoft has initiated a new convention with C#. Public entities begin with an uppercase letter, and private ones begin with a lowercase. Your program will still run and compile just fine if you ignore this convention, but it's worth adopting, because other C# programmers will expect the capitalization to provide a clue to whether a variable is public or private. I prefer the preceding p convention because it's recognized in pretty much every language I use.

The pName variable holds the value of the critter's name but is not directly accessible to the outside world. Instead, I will create some special methods that allow outside entities, such as another class or programmer, access to this variable.

Examining the Basic Design of a Property

A *property* is a pair of special methods, get() and set(), that control access to a private instance variable. Here's the essential property description for the name property:

```
public string name {

  get {
```

```
    return pName;
  } // end get

  set {
    pName = value;
  } // end set

} // end name property
```

The name property is declared as public because anybody can have access to the property. Properties have types because they are methods that control access to variables. In essence, the get() and set() methods of a property combine to create a "wrapper" around a local variable. A property should have the same type as the variable it represents. A property contains a get() method and a set() method. The get() method is used when a program wants to get the value of the property from the class, and the set() method is used when a program wants to set the value of the property.

> **Trap** The terminology here can sound crazy. You might think that the get() method would get a value for the property, but that's not the way it works. The get() method is used to send the property out to the rest of the world, and set() is used to send a value to the property from the outside. If you take the point of view of the program that uses your object, it all makes sense.

The get() and set() methods aren't exactly like other methods. They must always have the names *get* and *set,* and they exist only in the context of a property definition. Also, they do not have parameters like other methods.

Creating a get() Method

The get() method returns a value for a particular property. Therefore, a get() method always has a return statement. Usually, the get() method is very simple, as in this example, but it can become complex, especially if the property is not directly stored but is calculated from other elements.

Creating a set() Method

The set() method is designed to accept a potential value for a property and assign it to the private instance variable. The value that the calling program is trying to send to the program is stored in a special (undeclared) parameter named value. The simplest form of the set() method assigns value to the private instance variable your property surrounds. If you want a read–only property (one like area, described earlier, which can't be changed from the outside), simply omit the set() method.

Using Properties as Filters

The great thing about properties is that they give you error–checking capabilities. Suppose that there is a rule stating that critter names cannot be more than eight characters long (maybe they are DOS critters). If name was a public instance variable, it would be difficult to ensure that this eight–character rule is enforced. However, if name was a property, making sure that critter names always follow the rule would be easy. Take another look at the set() method I built for the Critter class:

```
set{
  if (value.Length > 8){
    Console.WriteLine("The name can't be more than 8 characters");
    pName = value;
```

```
    pName = pName.Substring(0,8);
    Console.WriteLine("Changing name to {0}", pName);
  } else {
    pName = value;
  } // end if
} // end set
```

The set() method examines the incoming value and checks whether it is more than eight characters. If it is, the set() method performs string manipulations to shorten the name (after informing the user of this action). If the incoming value is eight characters or fewer, it is assigned directly to the private variable.

In the Real World

The name restrictions in the following example are silly, but the concept holds true in serious programming. I wrote a program to simulate an air traffic control system. The **plane** object had a property to take a direction. I used a property to enforce that the direction value be between 0 and 360 because these are the legal direction values in degrees.

Also, because many kinds of input come in as strings, I've written properties that accept a string value and convert it into the type I need.

Properties enable you to wrap instance variables in a protective filter of the **get()** and **set()** methods so that you have more precise control over the input and output of these properties.

The same kind of technique can be used for any kind of filtering. Suppose that you want all your critter names to start with *The Honorable* or you want to refuse all names that don't include a *q*. No problem! Just use the set() method to designate these restrictions.

Making the Critter More Lifelike

You now know all you need to know to make the critter a lot of fun. By adding a few more variables, properties, and methods, you can re–create the critter featured at the beginning of this chapter. Remember that my original algorithm was to have a critter whose moods change according to the way it is treated. I wanted to achieve this effect as easily as possible.

Adding More Private Variables

I added a couple private variables to the critter, but not to make them properties (at least, not yet), because I wanted the user to interact with them indirectly. Here's the list of private variables and properties for the final Critter class:

```
class Critter {
  private string pName;
  private int pFull = 10;
  private int pHappy = 10;
  private int pAge = 0;

  public string Name {
    get {
      return pName;
```

```
    } // end get
    set{
      if (value.Length > 8){
        Console.WriteLine("The name can't be more than 8 characters");
        pName = value;
        pName = pName.Substring(0,8);
        Console.WriteLine("Changing name to {0}", pName);
      } else {
        pName = value;
      } // end if
    } // end set
  } // end name property
```

The name property and its associated private variable, pName, are identical to the version in the preceding section. However, I added a few new private variables. The pFull variable describes how hungry the critter is. If pFull is zero, the critter is hungry. You'll see how it works when you look at the new methods in the Critter class. The pHappy determines how happy the critter is. As with the pFull variable, a larger value is better. The pAge variable determines the critter's age.

Adding the Age() Method

To make the program interesting, I wanted to degrade the critter's situation slightly each time through the loop, so I added an age() method to the Critter class:

```
public void Age(){
  //handles aging the critter
  pAge++;
  pFull--;
  pHappy--;

  if (pFull < 3) {
    //if hungry, accelerate unhappiness
    pHappy--;
  } // end if
} // end age
```

The effects of aging on a critter are obvious. The age increases by 1, and fullness and happiness decrease by 1. As an added effect, I decided that hunger would accelerate unhappiness (it works that way with my kids, anyway), so if the critter is hungry, it quickly becomes unhappy.

The Age() method will be added to the Main loop so that the critter ages once per turn.

Adding the Eat() Method

The feed() method gives the user the opportunity to feed the critter. The digestive system of a critter takes little code to reproduce.

```
public void Eat(){
  pFull += 4;
} // end eat
```

This has the effect of stuffing the critter, by incrementing the value of pFull by 4. Recall that pFull += 4 is just like pFull = pFull + 4;.

Adding the Play() Method

The play() method is very similar to the Eat() method, but it changes the happiness level rather than the hunger level. For the sake of variety, I decided that the Play() method should not have quite as strong an effect on happiness as the Eat() method has on hunger.

```
public void Play(){
  pHappy += 3;
} // end play
```

To give your critter a different personality, you can alter the way the variables are changed. For example, the current design of the critter changes only one variable at a time. During one turn, you can play with a critter or feed it, but you can't do both. Because each attribute is decremented each time through the loop, the critter's happiness level goes down during the turn in which you feed it. Of course, you can change this by altering the feed() and play() methods.

Modifying the Talk() Method

The talk() method is the critter's primary feedback mechanism. Only by asking can you find out how the critter is doing. The talk() method provides all the information about the critter. Here's my version of the talk() method:

```
public string Talk(){
      string message;
      message = "The critter says: \n";

      if (pHappy > 5) {
        message += "  Hi!  My name is " + Name + "\n";
        message += "  I feel happy today! \n";
      } else if (pHappy > 2) {
        message += "  " + Name + " doesn't feel so good...";
      } else if (pHappy > 0) {
        message += "  " + Name + " is MAD...";
      } else {
        message += " ...nothing at all, but lays in a heap.";
      } // end if
      return message;
    } // end talk
```

I wanted the critter's behavior to indicate its status indirectly, so I had it give varying responses, based on its happiness variable. Although a direct readout of the critter's characteristics would have been more informative, no other type of pet makes it this convenient—neither should the critter. (Of course, you can change this if you like.) I didn't respond directly to hunger because happiness is affected by hunger, but you can make your critter act the way you want.

Making Changes in the Main Class

The improvements in the Critter class make the Main class much easier and more powerful. The menu method itself doesn't change at all, but you can finally add responses to all the menu items:

```
class Menu {
  static void Main(string[] args) {
    bool keepGoing = true;
    int choice;
    Critter myCritter = new Critter();
    myCritter.Name = "George";
```

```
  while (keepGoing){
    myCritter.age();
    choice = showMenu();
    switch (choice){
      case 0:
        keepGoing = false;
        break;
      case 1:
        Console.WriteLine(myCritter.Talk());
        break;
      case 2:
        myCritter.Eat();
        Console.WriteLine ("You have fed the critter");
        break;
      case 3:
        myCritter.Play();
        Console.WriteLine("You have played with the critter");
        break;
      case 4:
        Console.WriteLine("Current name: {0}", myCritter.Name);
        Console.Write("Change name to: ");
        myCritter.Name = Console.ReadLine();
        break;
      default:
        Console.WriteLine("That was not a valid input");
        break;
    } // end switch
  } // end while loop
} // end main
```

The main program still reports the user's actions, but all the real work is delegated to methods of the Critter class. Each time through the loop, the program calls the critter's Age() method, and then each menu item results in a call to the appropriate method of the Critter class. If the user wants to feed the critter, the menu calls the critter's Eat() method. If the user wants to play with the critter, the menu calls the critter's Play() method. The code that deals with user interaction happens in the Menu class, and the code that deals with the details of the critter's behavior goes in the Critter class. Encapsulation is great.

Summary

In this chapter you looked at several forms of encapsulation. You learned about dividing a program into manageable pieces, using several strategies. You built methods to encapsulate code fragments and saw how those methods can receive values through parameters and output values with the return statement. You learned how to build basic classes to encapsulate data and methods and how to make an instance of a class. You added methods and properties to your classes and learned some basic exception handling. With mastery of these skills, you are no longer simply a programmer. You are well on your way to becoming an object–oriented programmer.

Challenges

- Modify the program so that it shows the critter's age. Give a special reward to the owner who can keep a critter happy for a certain number of turns.
- Add another method to the critter, such as **tickle()** or **sleep()**. Determine how this method would relate to the critter's variables. Don't forget to change the menu to take advantage of your new method.

- Add a cash variable. Maybe decrease the amount of money in your account each time the user feeds the critter, but award cash for longevity milestones.
- Create the critter's capability to do tricks. For example, if the critter reaches a certain level of happiness, a new trick or method is made available. You might combine this with a cash variable so a critter that can do fancy tricks can earn his own keep. (If only they paid cats for scratching couches...)
- Modify the program to make a virtual version of one of your pets. (One person I know sent out object–oriented birth announcements to his programming friends!)

Chapter 5: Constructors, Inheritance, and Polymorphism: The Snowball Fight

With the ability to create custom objects, you begin your adventure into object−oriented programming (OOP). However, you have yet to learn some other very important characteristics of objects. In this chapter you will explore the essential techniques of OOP and learn how

- To write a constructor to customize the way a class is instantiated
- To overload constructors for flexibility
- Inheritance is used to reuse code
- Polymorphism is used in OOP

Introducing the Snowball Fight

With surprisingly little work, you can use objects to model complex behavior. In this chapter you will make a model of a snowball fight. Each player is an object, and the menu system is a third object. Figures 5.1, 5.2, and 5.3 illustrate the snowball fight in progress.

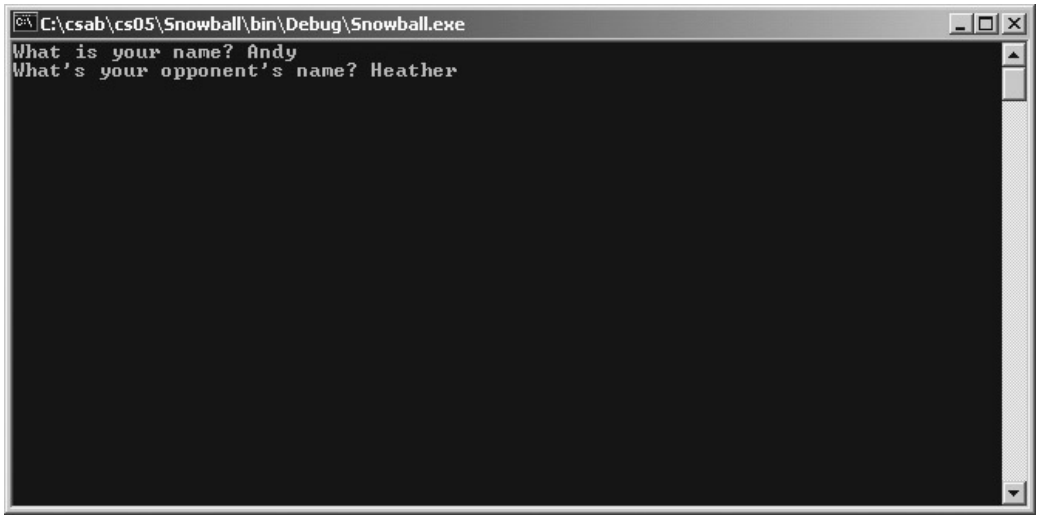


Figure 5.1: Begin by entering the player names, which are important for the play−by−play description of the game.

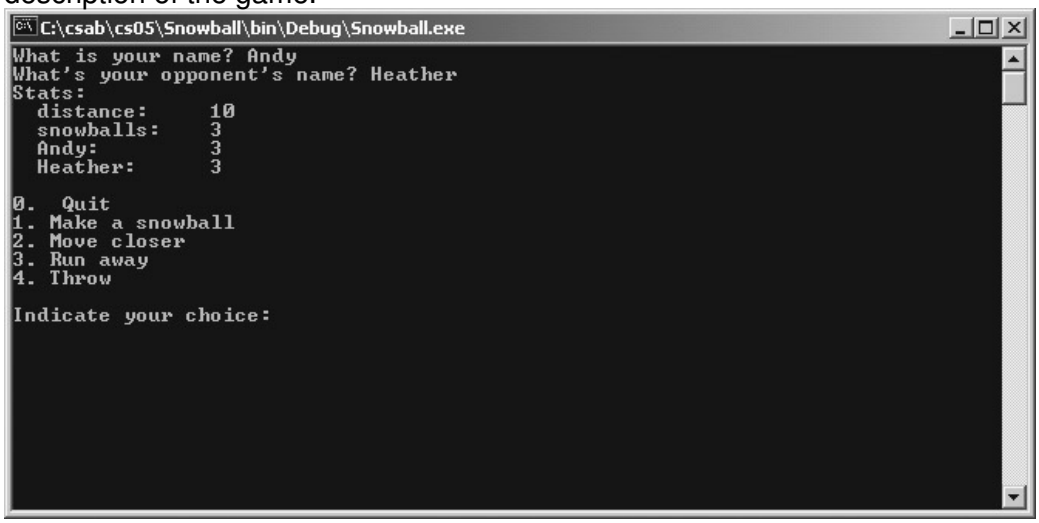


Figure 5.2: Players have a limited number of snowballs and are more likely to hit their target when they are close to it.

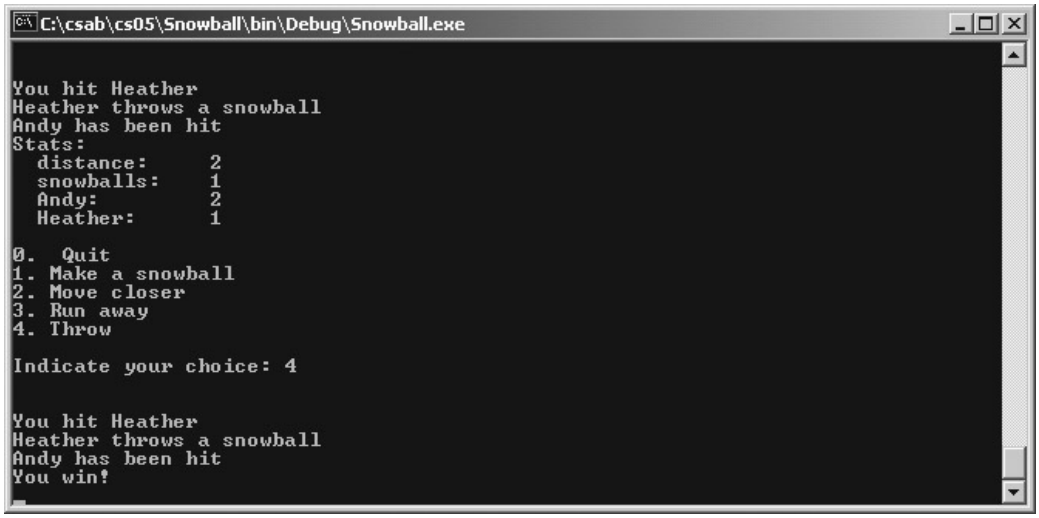


Figure 5.3: With a good strategy, you can beat the computer much of the time, but not always.

The design of the snowball fight is simple. Both the player and the opponent are custom classes that have a lot in common. They have three properties: name, snowballs, and strength. Each player starts with three snowballs and, to win, has to make more during the fight. Each player also begins with three lives. When a player has been hit three times, that player loses the fight. The computer opponent is like the human player but has features that enable it to automatically play against the user.

Inheritance and Encapsulation

To make the snowball fight and other programs in the book, you must learn a few tricks about building classes. To illustrate these new concepts, I'll build a few more critters like the ones in Chapter 4, "Objects and Encapsulation: The Critter Program." The basic critter is interesting, but what if you want to make new kinds of critters that share the same behavior but exhibit different characteristics? For example, you might want to make a new kind of critter that is grumpy.

When programmers began using objects, they quickly realized the importance of being able to make changes to objects. Computer scientists (who love to obfuscate simple ideas by using techie terms) think that objects should support *inheritance* and *polymorphism*. These fancy words describe some simple but important ideas. Inheritance works much like genetics. Objects can have children and grandchildren, and an object's descendants will inherit traits from the parents and grandparents. *Polymorphism* means that an object can have the same kind of behavior as its relatives but can implement that behavior differently. Don't worry if these explanations leave you cold for now. I just want you to have a bird's–eye view of these concepts before digging into specific examples later in this chapter.

Creating a Constructor

Imagine that you want to write a program using several critters. Each critter could have a different name and different characteristics (perhaps different starting values for hunger and happiness). Using the Critter class from Chapter 4, you could do it, but you'd have to create each critter in several steps. First, you'd have to create the critter, and then you'd need another statement of code to modify each characteristic. It would look something like this:

```
Critter myCritter = new Critter();
```

```
myCritter.name = "alpha";
myCritter.age = 10;
```

Although this is not difficult, a more convenient way to create a critter would enable you to initialize all its values at the same time. You could set up a critter with a line like this:

```
Critter myCritter = new Critter("alpha", 10, 9, 0);
```

The critter automatically assumes the name *alpha*, a happiness level of 10, a hunger level of 9, and an age of 0. (The age of 0 means that it will age after the first turn) You can apply a set of parameters whenever you create an instance of the critter class. Classes (such as the Critter) can have a special method called a *constructor*. A class's constructor is a special method that is used to help create the critter. The constructor is automatically called whenever you create an instance of the class when using the new keyword.

Adding a Constructor to the Critter Class

I took the Critter class from Chapter 4 and added a constructor to it:

```
using System;

namespace CritterConstructor
{
  /// <summary>
  /// Critter with a constructor
  /// </summary>

    public class Critter {
      // your basic critter

      //instance variables
      private string pName;
      private int pFull = 10;
      private int pHappy = 10;
      private int pAge = 0;

      //constructor
      public Critter(string theName, int fullness, int happiness,
int theAge){
        name = theName;
        pFull = fullness;
        pHappy = happiness;
        pAge = theAge;
      } // end constructor

      public string name
      public string talk() ...
      public void age() ...
      public void play() ...
      public void eat() ...
    } // end class

} // end namespace
```

I minimized all the properties and methods so that you can focus on the new part of the critter. *Constructors* are special methods that have the same name as the class. A constructor is usually public, and you do not have to specify a return value because the constructor will return an instance of the class as its value. Whenever you make an instance of a class (remember, the class is a

recipe, and instances are the cookies), the computer looks for a constructor. The constructor usually has special instructions for getting the class started. In this case, I added a constructor for the Critter class that accepts four parameters. Each parameter is mapped to a specific instance variable. Constructors are typically used to initialize instance variables, even in more complex classes. The other nice thing about a constructor is that you know that any code inside a constructor will happen as soon as the object is created. Therefore, any code you want to run at the beginning of the class's life span should be written in the constructor. Constructors can have parameters just like any other method.

The Critter class does not have a Main() method. In any given namespace, it generally makes sense for only one class to have a Main() method. However, your project can (and usually will) have many classes and several instances of each class. The Critter class is not intended to be an executable program, so I will use it inside other classes that do have a Main() method. When you look at the files created by the .NET compiler, the programs that have a Main() method usually have an associated .exe (executable) file. Those that do not have a Main() method are compiled into dynamic linked libraries (.dll files). You might have heard that the only difference between an .exe file and a .dll file is that an executable (.exe) file has a Main() method and a dynamically linked library (dll) doesn't. This is a generalization (and not completely true). Important internal differences exist between these files, but for the purposes of this book, this is a reasonable simplification.

Creating the CritViewer Class

If you type in the Critter class from the preceding section and attempt to run it, it will compile, but it won't run. Often, you make classes that are meant to be used as parts inside other classes. A gas tank, for example, is an important part, or class, of the automobile. Gas tanks have parts and require assembly. When a gas tank is finished, it is sent to the auto assembly plant and installed. The gas tank is a class, but you can't drive it because it's designed to be part of a larger assembly. The program itself is one object (like the car), but it usually comprises constituent objects. Only the primary object needs a Main() method.

The Critter class will not stand on its own, so you need a container class to demonstrate the critter's capabilities. In Chapter 4, you use a menu class to do this. Here, you use a much simpler program to contain the critter so that you can concentrate on new ways of building objects. The Critter Constructor program featured in Figure 5.4 demonstrates a version of the Critter program, with one class for a viewer and the modified Critter class with a constructor.

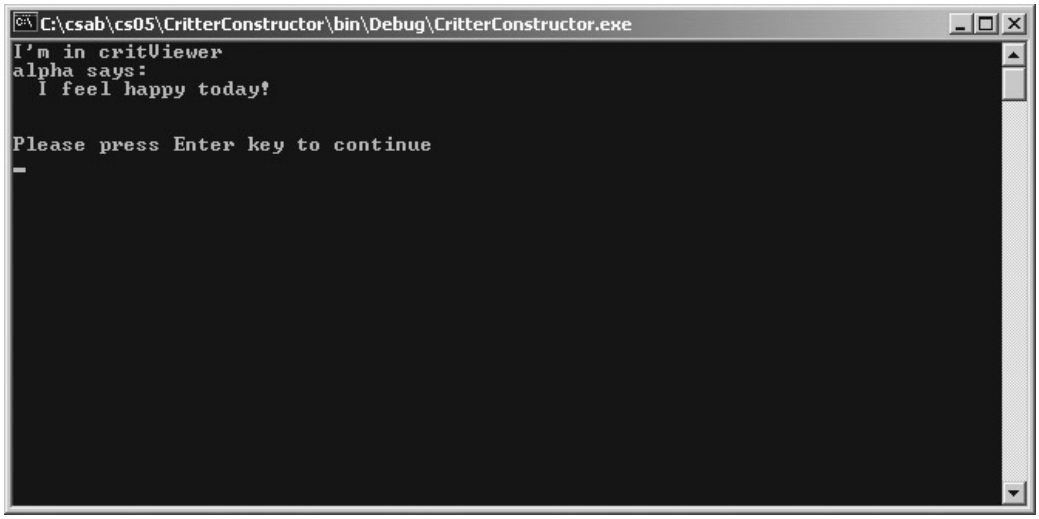


Figure 5.4: The critter viewer demonstrates the basic functionality of the critter.

Hint

Building a simple container to demonstrate and test a new class is a common strategy of object–oriented programmers. Because an object is encapsulated, it should work as well in one program as in another. You can create very straightforward programs to test your objects before you use the objects in a more complex program. With this technique, you are more likely to isolate errors in your custom classes before you add them to complex assemblies, where more can go wrong.

To demonstrate my new critter, I'll build a CritterViewer class, which is reasonably simple:

```
using System;

namespace CritterConstructor
{
  /// <summary>
  /// CritViewer is a simple class designed simply to hold a critter
  /// Demonstrates self-instantiation
  /// Andy Harris, 12/21/01
  /// </summary>
  class CritViewer
  {
    static void Main(string[] args)
    {
      // the main method simply creates an instance of the
      /// critviewer object
      CritViewer cv = new CritViewer();
    } // end main

    //This next method is the constructor for CritViewer
    public CritViewer(){
      Critter myCritter = new Critter("alpha", 10, 10, 0);
      Console.WriteLine("I'm in critViewer");
      Console.WriteLine(myCritter.talk());

      Console.WriteLine();
      Console.WriteLine("Please press Enter key to continue");
      Console.ReadLine();
    } // end constructor
  } // end CritViewer Class
} // end namespace
```

Note the simplicity of the Main() method in this code:

```
static void Main(string[] args)
    {
      // the main method simply creates an instance of the
      /// critviewer object
      CritViewer cv = new CritViewer();
    } // end main
```

Reviewing the Static Keyword

So far in this book, most of the code appears in the Main() method of the program. Although this approach is fine for these simple programs, it has limitations because the Main() method must be declared a *static* method. The keyword static in the preceding code means that the Main() method can be called before the class exists. A static method can be called without requiring an instance of the class. For example, most of the methods in the Convert class in Chapter 2, "Branching and Operators: The Math Game," are static methods. You don't have to create an instance of the Convert class to use its methods. Likewise, the WriteLine() method of the Console class is static.

Static methods can be useful, but they have a serious limitation: Static methods cannot refer to instance variables because the instance variables have meaning only in an instance. The Main() method must be declared static, because it is the first entry into the program from the operating system. To avoid some of these limitations of static methods, the Main() method is usually much simpler than it has been in the examples you have seen so far in this book.

That sounds complicated, but it isn't. The keyword static can also be read as *class–level*. The keyword instance in the preceding code refers to *instance–level*. Recall the cookie recipe analogy. A recipe is a class, and the cookies are instances of that class. A static method belongs to the entire class, not to a specific instance. In other words, a static method is a method that can be applied to the class, but not necessarily to the instances of that class. A recipe class may have copy and e–mail methods. An instance of a cookie is not required to invoke the class–level methods. It doesn't make sense for e–mailing to belong to cookies (the instances of the class). (Besides, cookie dough is very hard on floppy drives.) E–mailing is a method of the recipe (the class itself), so Cookie.email() would most likely be a static method. The e–mail method is not concerned with actual cookie instances, so it makes sense that the static method would not have access to the details of individual cookies. Because you are e–mailing the recipe, not a cookie, the e–mail method shouldn't have access to the bitesMissing property of a cookie because bitesMissing is an instance–level property (as most properties are).

Calling a Constructor from the Main() Method

The Main() method has to be a static method because it is called from the operating system. When you run a C# program, the first thing the operating system does is look for a Main() method. That Main() method runs before any specific classes are instantiated. Usually, it calls a constructor or two to get things started. A Main() method rarely contains much more than one call to a constructor because the static limitations can get in the way. Instead, the Main() method usually creates an instance of a class or two. You might be surprised to see which class the CritViewer's Main() method creates:

```
CritViewer cv = new CritViewer();
```

The Main() method of the CritViewer class creates an instance of the CritViewer class! This idea might seem like the work of the department of redundancy department, but it makes sense in terms of static methods. Main() is a static method, which means that it runs before an instance of the CritViewer object occurs. Because I really want an instance of the object here, I use the Main() method to create an instance of the class. Objects with a Main() method often utilize this technique to pull themselves into existence. When you run a program that contains an object with a Main() method, that Main() method runs before any other code executes. The main method calls the class' s constructor, which finishes creating the class.

Trick If more than one class has a Main() method, you can set the properties of the project in the project window to determine which object's Main() method will start the program.

Examining CritViewer's Constructor

The rest of the work in the CritViewer class occurs in its constructor:

```
//This next method is the constructor for CritViewer
    public CritViewer(){
      Critter myCritter = new Critter("alpha", 10, 10, 0);
      Console.WriteLine("I'm in critViewer");
      Console.WriteLine(myCritter.talk());
```

```
    Console.WriteLine();
    Console.WriteLine("Please press Enter key to continue");
    Console.ReadLine();
  } // end constructor
```

Constructors are instance–level because they create an instance of an object. The constructor of the critter viewer creates an instance of the Critter class. I took advantage of the critter's newfound constructor capabilities. Building the critter with this four–parameter constructor is convenient because it saves you a lot of typing time.

Working with Multiple Files

Until now, all the source code in your programs has existed in a single file on the disk. This is fine for small programs but becomes unwieldy when your programs are longer. After you start building programs with multiple classes, you should open a new file for every class. The default file that loads in the editor when you start a program already has a Main() method. You usually use that program as the container class to hold your other classes. Figure 5.5 demonstrates how to create a new class in the Visual Studio editor.

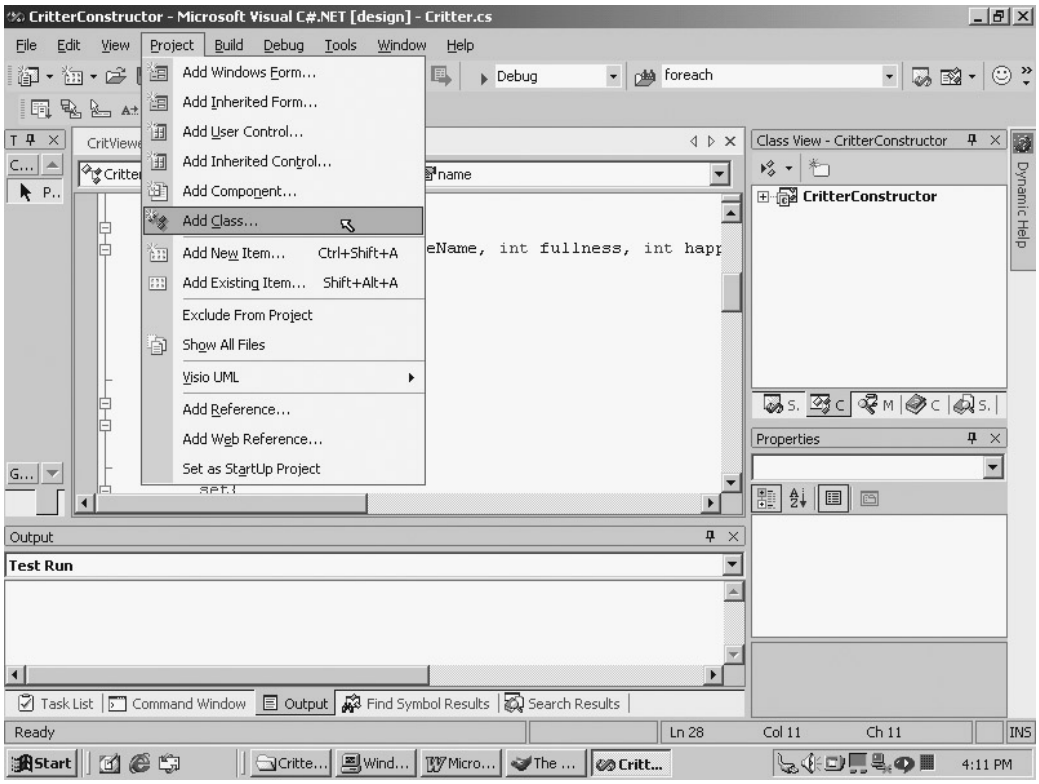


Figure 5.5: To create a new class, choose Add Class from the Project menu, then click Class.

The Visual Studio IDE enables you to have many files open at the same time. This can be very handy when your programs become complex. As a default, the new class will belong to the same namespace as your existing classes because all the classes in your project are meant to work together.

The IDE also has a class viewer feature for looking at all the parts of your object. The Class View, illustrated in Figure 5.6, is usually available in the right segment of the editor.

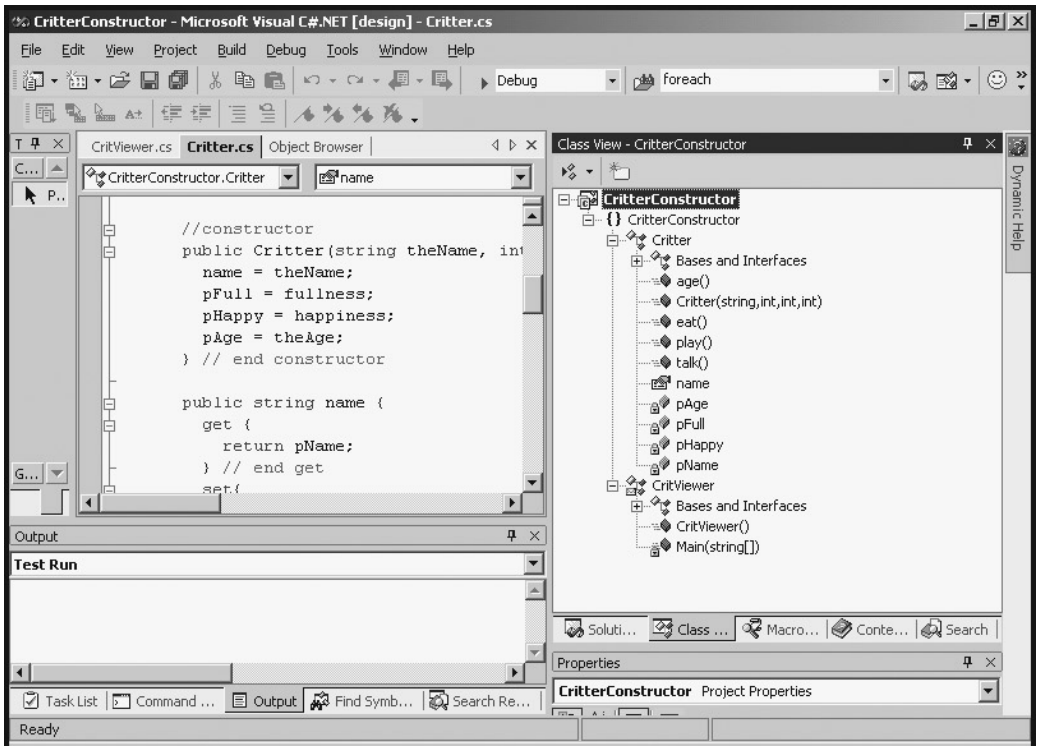


Figure 5.6: The Class View is used to navigate the entire project.

This Class View is a specialized object browser containing the objects in your project. (You can also view your project in the object browser window if you like, but the Class View is usually more convenient for this purpose.) You can expand the various elements of your project to see every property and method of your custom objects. When you double–click a property or method in this menu, you are taken directly to the code related to that member. You can use the Class View as a menu system for your code, giving you an easy way to jump to whatever part of the code you want to view.

Overloading Constructors

Constructors enable you to send parameters when you build an object, but sometimes you don't want to send any parameters. You can create an object in one of several ways. Classes can have more than one constructor, which makes them even more flexible. If you have more than one constructor, it is known as *overloading* your constructors.

Viewing the Improved Critter Class

You can make another version of the Critter class that has several constructors with different sets of parameters. Take a look at this version of the critter to see how it works. I'm showing only the constructors in this code listing because nothing else changes.

```
using System;

namespace CritOver
{
  /// <summary>
  /// Critter class showing overloaded constructors
  /// Andy Harris, 12/21/01
  /// </summary>
  public class Critter {
```

```
    // your basic critter
    //instance variables
    private string pName;
    private int pFull = 10;
    private int pHappy = 10;
    private int pAge = 0;
    //overloaded constructors
    public Critter(string theName, int fullness, int happiness,
int theAge){
      name = theName;
      pFull = fullness;
      pHappy = happiness;
      pAge = theAge;
    } // end constructor
    public Critter(string theName){
      name = theName;
      pFull = 10;
      pHappy = 10;
      pAge = 0;
    } // end constructor
    public Critter(){
      name = "";
      pFull = 10;
      pHappy = 10;
      pAge = 0;
    } // end constructor

    public string name ...
    public string talk ...
    public void age() ...
    public void play() ...
    public void eat() ...

  } // end class
```

You can see that this version of the Critter class has three constructors. You can have as many constructors as you like, as long as each constructor accepts a different number and type of parameters. The number and type of values in a parameter are called a *parameter signature*. When you create an instance of the Critter class, the computer searches the class for a constructor with the parameter signature you specify. When you have multiple constructors with different signatures, you have *overloaded constructors*. Overloaded means that you have supplied more than one way to do something. You can overload any method you like, not just constructors, although constructors are the most common method to overload. Nearly every class can benefit from a variety of invocation techniques, and overloaded constructors provide this flexibility.

Trap When determining whether a parameter signature is unique, the compiler considers only the *type* and *number* of arguments, not their names. For example, new Critter("Buddy Holly"); looks for a constructor that takes one string as an argument. If you had two constuctors in your object: public Critter(String name) {} public Critter(String description) {} The compiler would be confused as to which "Critter" you were trying to create because they look the same to the compiler.

Adding Polymorphism to Your Objects

One of the magic techniques I promised to teach you at the beginning of the chapter is *polymorphism*.

Polymorphism, in a nutshell, means that a class can do the same thing in different ways. Methods with different parameter signatures offer one form of polymorphism. Polymorphism in classes means that several classes can have the same method, but that method happens differently in each class. For example, a chainsaw, car, and motorcycle all have a start() method that starts up the engine. However, the underlying mechanics of starting a car with an electric motor activated by a key are very different from the way you kick–start a motorcycle or pull the lanyard of a chainsaw. Each object has a start() method, but the start() method is implemented differently in each type of object.

Another form of polymorphism is the ability to create things in more than one way. You can use overloaded methods and constructors to create a form of polymorphism in your classes. Imagine that you have a method that returns the square of a real number. That method could look like this:

```
public double getSquare(double theNumber){
  return(theNumber * theNumber);
} // end getSquare
```

You might want another version that works on integers. You could overload the method with an integer version, as shown in the following code:

```
public double getSquare(int theNumber){
  return (int)(theNumber * theNumber);
} // end getSquare
```

The version of the method that accepts an integer works the same as the one that accepts a double. Both return the same value, but they work on different types of data. You can write a program that sends an int or a double to the function without worrying about the type of input. The program works well on different types of data and automatically corrects for whatever kind of input it gets.

To take this idea to its extreme, you would need several versions of the method, one for each of the main types of data. If you look at many methods in the .NET system classes, you'll see that they do exactly this. For example, there are 19 versions of the Console.WriteLine() method, which is why it seems as if you can send any kind of data to the WriteLine() method. The method has been so overloaded that the console object can guess how to write to the screen nearly anything you want to pass to it. Polymorphism and method overloading make your classes easier to use because a programmer using your class has choices for how to create your class. Polymorphism can also eliminate certain kinds of errors because your class can anticipate data being sent in an inappropriate format and automatically change the information to the format it needs.

Modifying the Critter Viewer in CritOver to Demonstrate Overloaded Constructors

After you add new features to a class, you improve your container class to test those new features. For the Critter Over program, I modified the CritViewer class so that it would make three versions of the critter, each with a different constructor:

```
using System;

namespace CritOver
{
  /// <summary>
  /// Demonstrates overloaded constructors
  /// </summary>
```

```
  class CritViewer
  {
    static void Main(string[] args)
    {
      CritViewer cv = new CritViewer();
    } // end main

    public CritViewer(){
      //Create some critters
      Critter alpha = new Critter("alpha", 10, 10, 0);
      Critter beta = new Critter("beta");
      Critter charlie = new Critter();

      //Make 'em talk
      Console.WriteLine(alpha.talk());
      Console.WriteLine(beta.talk());
      Console.WriteLine(charlie.talk());
      Console.ReadLine();
    } // end constructor
  } // end CritViewer class
} // end namespace
```

The program works correctly, as you can see in Figure 5.7.

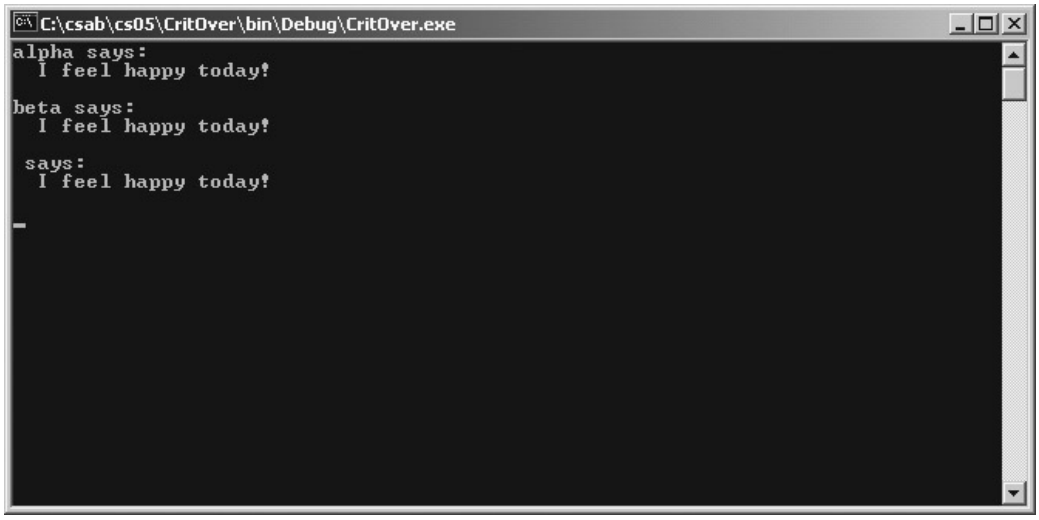


Figure 5.7: Although not apparent from the output, each critter uses a different constructor. The last one begins with a blank name because it was called with no parameters.

Using Inheritance to Make New Classes

The other major concept I introduced at the beginning of this chapter is *inheritance*. Like many computing words, *inheritance* is borrowed from the nontechnical world, but computing types added new significance to the term. The basic idea of inheritance in object–oriented programming is similar to the genetic meanings of *inheritance*: You have characteristics of your parents and your grandparents. Objects also have a family tree. Nearly every object in C# has exactly one parent. That parent might have a parent, and this class might also have a parent. A class can inherit characteristics from each member of its family tree, just as you might have your grandmother's nose and your mother's eyes. Inheritance in C# is much simpler than in genetics, however, because each class has only one parent.

Trap Some OOP languages (such as C++) allow an object to have more than one parent class, but multiple inheritance causes many more problems than it solves. C# uses an inheritance model that is simpler and less prone to error.

Creating a Class to View the Clone

To illustrate inheritance, I'll show you the simplest example of inheritance I know.

Figure 5.8 illustrates the output of another version of the Critter program. From the output, this looks very much like the other programs you have seen in this chapter. However, when you look at the code that creates the Clone class, you'll find a surprise.

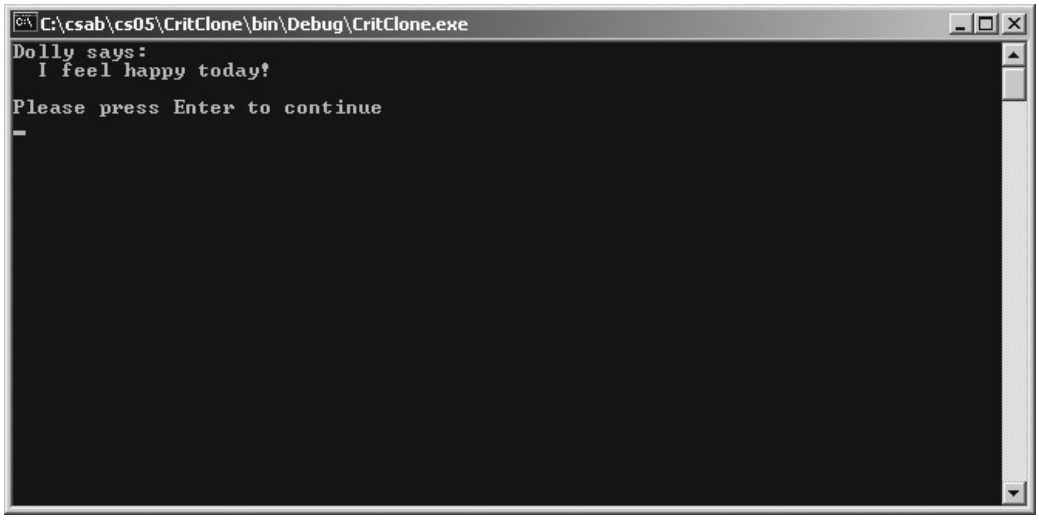


Figure 5.8: The output looks very familiar, but Dolly is actually a clone!
Note that the code for CritViewer is also familiar:

```
using System;

namespace CritClone
{
  /// <summary>
  /// Another Critter Viewer
  /// This one demonstrates inheritance by making a clone
  /// Andy Harris, 12/21/01
  /// </summary>
  class CritViewer
  {
    static void Main(string[] args)
    {
      CritViewer cv = new CritViewer();
    } // end main

    public CritViewer(){
      Clone myClone = new Clone();
      myClone.name = "Dolly";
      Console.WriteLine(myClone.talk());

      Console.WriteLine("Please press Enter to continue");
      Console.ReadLine();
    } // end constructor
  } // end CritViewer class
} // end namespace
```

This program never directly invokes the Critter class. Instead, it makes an instance of the Clone class. The clone acts much like a critter. When you type it into the editor, you get the same list of properties as when you are working directly with a critter. Also, the clone has a talk() method that works just like the Critter class.

Creating the Critter Class

From all the evidence, a clone would seem to be almost exactly like a critter. Here's the code for the clone:

```
using System;

namespace CritClone
{
  /// <summary>
  /// The Clone is a very simple class
  /// Illustrates basic inheritance
  /// Andy Harris, 12/21/01
  /// </summary>
  public class Clone: Critter
  {
    // there's nothing here!
  } // end class
} // end namespace
```

The most startling part of the Clone class is what *isn't* there. The clone has no properties, methods, or constructors, yet it acts as if it has them. The Critter Viewer program uses the talk() method and changes the name property. The Clone class acts much like the Critter class, but I didn't have to rewrite the critter's characteristics. The key is in the way the class is derived. Look at this line of code:

```
public class Clone: Critter
```

This line defines Clone, but the colon followed by a class name indicates that the Clone is *derived* from Critter. In other words, I am starting from the Critter class, so the Clone class will start with all the characteristics of the Critter class. Without writing a single new line of code, I've made a new class related to an existing class.

Trap Having an exact duplicate of an existing class is pointless without modification. I wanted to show you the concept of inheritance with a clear example. In the rest of this chapter you will learn how to make modifications to your copy so that it has new behavior, methods, and properties.

Because Clone is derived from Critter, it inherits all of Critter's characteristics. Clone has a talk() method because Critter does, and Clone is derived from Critter. Clone also has access to all of Critter's other properties and methods.

Improving an Existing Class

Most of the time, you don't need to build an object completely from the ground up. Usually, your object can be based on another object. The ability to make one class based on another class is the foundation of inheritance. For example, imagine that you are a car designer, and you have defined a car factory that produces a standard car. If you want to build taxis and police cars, you don't have to start from scratch because your new car can inherit all the basic characteristics (wheels, doors, engines) of its parent class, the standard car. To make a taxi class, you start with a car class and add the characteristics that make it a taxi (a horn that blows constantly and the capability to splash pedestrians are requisite features).

Most projects involve using classes in as many as three ways. You use existing classes, such as the Console and Convert classes. You sometimes make entirely new objects, as I did with the Critter. Also, you frequently modify existing classes and add new capabilities to them.

Introducing the Glitter Critter

To illustrate how a new class can modify an existing class, look at the glitter critter. This is a variation of the Critter class that has a new method, shine(). You can see the glitter critter in all its glory in Figure 5.9.

Figure 5.9: The glitter critter is like normal critters, but it has a shine() method.

The GlitterCritter class also supports all the constructors of the Critter class. Because most of the behavior already belongs to the Critter class, the GlitterCritter code is dedicated mainly to the new elements of this new class:

```
using System;

namespace GlitCrit
{
  /// <summary>
  /// New version of Critter that adds a "shine" method
  /// </summary>
  public class GlitterCritter : Critter
  {
    //overloaded constructors map to base (Critter) constructors
    public GlitterCritter(): base(){ }
    public GlitterCritter(string name): base(name){ }
    public GlitterCritter(string theName, int fullness, int happiness,
int theAge)
           : base(theName, fullness, happiness, theAge){ }

    //new shine method
    public void shine(){
      Console.WriteLine(name + " shines beautifully.");
    } // end shine

  } // end GlitterCritter
} // end namespace
```

The GlitterCritter sports three constructors and a method. It inherits everything else from its parent class, the Critter.

The joy of inheritance is that you don't have to keep reinventing the wheel. You can often base your class on an existing class and have automatic use of all its methods. If the class you are deriving from is inherited from another class, you have access to all the properties and methods of the grandparent class as well. Returning to the car example, you can have a sedan class, which is a standard car. You can derive a heavy−duty sedan class with the same features as a sedan but modifications for the functions of a commercial vehicle. You can have a taxi class with features exclusive to taxis (the meter), features of the heavy−duty sedan class (a special suspension), and many features of the original sedan class (steering wheel, doors, and so on). After you build the heavy−duty sedan class, making new commercial vehicles is easier because all you have to worry about are those characteristics that distinguish a class from its parent.

In the Real World

Part of being an object−oriented programmer is knowing the humor. You're now ready for the classic object−oriented programming joke:

How many object−oriented programmers does it take to change a light bulb?

None! You should inherit the change() *method from the light bulb's parent class!*

Calling the Base Class's Constructors

The code for the GlitterCritter class has several constructors that are very simple, but not much like the other methods you've come to know. For example, the default (no−parameters) constructor looks like this:

```
public GlitterCritter(): base(){ }
```

The colon after the constructor name lets you specify which constructor of the base (parent) class you want to call. In this case, if the user decides to instantiate GlitterCritter with no parameters, it calls the Critter class constructor, also with no parameters. The constructor has a pair of braces so that you can put code in it. However, inherited constructors often don't need much code because most of the building is done in the parent class. You can often leave the constructor code blank in an inherited constructor. The only time this isn't true is when you need to initialize a value or property that belongs to the child class but not to its parent. I added other constructors to the GlitterCritter class. They are very similar to the no−parameters version:

```
public GlitterCritter(string name): base(name){ }

public GlitterCritter(string theName, int fullness,
       int happiness, int theAge)
       : base(theName, fullness, happiness, theAge){ }
```

Each constructor must have a different parameter signature. For the GlitterCritter, I decided to use exactly the same types of parameters as the original Critter, so I had each constructor map to the similar constructor of Critter. I didn't need additional code in the GlitterCritter's constructors, so I left them blank for now.

Adding Methods to a New Class

The remaining feature of the GlitterCritter class is the new method, shine(). Creating a new method in an inherited class is easy. Usually, you create the method just as you did in the original class. However, if the method existed in the base class, you must think more carefully about what you want your method to do.

Changing the Critter Viewer Again

Again, I modified the critter viewer to test my new critter's behavior. Here's the new version of the critter viewer:

```
using System;

namespace GlitCrit
{
  /// <summary>
  /// Another Critter Viewer
  /// Adding capabilities to an inherited class
  /// Andy Harris, 12/21/01
  /// </summary>

  class CritViewer {
    static void Main(string[] args) {
      CritViewer cv = new CritViewer();
    } // end main

    public CritViewer(){
      GlitterCritter gc = new GlitterCritter("Sparky");
      gc.shine();
      Console.WriteLine(gc.talk());

      Console.WriteLine("Please press Enter to continue");
      Console.ReadLine();
    } // end constructor
  } // end CritViewer class
} // end namespace
```

The only new code in this program invokes the shine() method of the glitter critter.

Using Polymorphism to Alter a Class's Behavior

Adding new methods to an inherited class is easy, but sometimes you don't want to add a new behavior as much as modify an existing behavior. For example, the critter is a pleasant creature, but suppose that you want (for some bizarre reason) to make a grumpier critter. This bitter critter should have a talk() method, but that method should not be just like the standard cheerful Critter talk. The bitter critter's talk() method should have the same purpose but should be written differently.

This situation is exactly what polymorphism is meant to solve.

When your new class has a method, and its parent has the same method with the same signature, you create an opportunity for confusion. If both Critter and BitterCritter have a talk() method, you must indicate which version should be invoked. C# allows you to specify that you wish to override a method of a parent class. The technique, called *method overriding*, also prevents you from overriding an existing method without knowing that you're doing it. I'll describe these mechanisms

next as you use polymorphism to build the bitter critter.

A few changes are necessary to add polymorphism to a class. At its simplest, all you do is make a class with the same method name as one of the methods of its parent class:

```
/// <summary>
/// BitterCritter
/// Designed to demonstrate polymorphism
/// </summary>
public class BitterCritter: Critter
{
  //note it always starts out ticked off
  public BitterCritter():base("", 2, 2, 0){
  } // end basic constructor

  public BitterCritter(string name): base(name, 2, 2, 0){
  } // end one string constructor

  public new string talk(){
    string message = name + " glowers moodily. \n";
    message += base.talk();
    return message;
  } // end talk method

} // end bitterCritter
```

I made a few changes to the constructors so that the bitter critter always starts off hungry and angry. I also added the talk() method. To specify that I intend the talk() method to override an existing method, I added the new keyword to the method call. The new keyword indicates that this new version of the talk() method is the one that should take precedence. If you want to call the talk() method of the parent object, use the base.talk() syntax.

In the Real World

Polymorphism entails more than what I've described in this section, but you won't need to worry much about it until you get into specific types of situations (for example, casting an object into its parent's type). If the new keyword does not provide the behavior you need, look up the combination *virtual* and *overrides* in MSDN help.

Creating the Snowball Fight

The snowball fight will be easy to build now that you have the object–oriented principles within your reach. The program has only three classes: the menu, the human player, and the robot player.

> **Hint** When you are designing a program, you often start by thinking about the main elements of the program. In this case, it is simple to see that the three entities in this game are the two players and the interface, which will handle the program logic and the user input. Seeing the objects you need to create is not quite as easy, so sketch out your thoughts on paper before you start programming.

The Fighter class is the most important part of this game because it forms the basis of both the human and robot players. When I first designed the program, I wondered whether both the player and the robot should be the same object. Even though that turned out not to be the case, the robot

fighter is based on the human fighter. The interactions between the human and robot fighters form the foundation of the program. I started by building the human fighter. After I had it working well, I extended the fighter to make the robot fighter. As I was building these two objects, I used the main menu program to test my objects constantly and make sure that each fighter was acting as I expected. When I was comfortable that the two player objects were working correctly, I added scoring and end–of–game situations.

Building the Fighter

The Fighter class is the heart of this game because it represents the human player. Because the robot fighter is derived from the Fighter class, both classes share essential characteristics. For this reason, I had to design the Fighter class to be very flexible so that it could work well as a human or robot player.

Setting Up the Basic Fighter

When you are looking at a new class, look at the instance variables. The instance variables often describe the most important data in the class. This data is so important that it is often encapsulated into properties. The first part of the Fighter class's code creates instance variables and properties to handle strength, snowballs, and name:

```
using System;

namespace Snowball
{
  /// <summary>
  /// Basic Snowball fighter
  /// </summary>
  public class Fighter
  {
    //instance variables
    private int pStrength;
    private int pSnowballs;
    private string pName;

    //properties
    public int strength {
      get {
        return pStrength;
      } // end get
      set {
        pStrength = value;
      } // end set
    } // end strength

    public int snowballs {
      get {
        return pSnowballs;
      } // end get
      set {
        pSnowballs = value;
      } // end set
    } // end snowballs

   public string name {
     get {
       return pName;
     } // end get
     set {
```

```
        pName = value;
      } // end set
    } // end name
```

The code is simple. The Fighter class has a name property, which is a string. The name is important in this program because there is no graphic interface. Each player must have a distinctive name, or the player will have difficulty figuring out what is happening. The fighter also has two numeric properties. The snowballs property determines how many snowballs the player currently has stockpiled. The fighter won't be able to throw if he doesn't have any snowballs. (It sounds obvious, but this is the type of behavior you have to write in later.) The strength property describes how many hits are left before this fighter loses the game.

Writing the Fighter Constructor

The constructor for the fighter simply initializes the instance variables. The name is accepted as a parameter to the constructor. Although I thought about overloading the constructor, it wasn't necessary in this situation because there was no need for other constructors.

Trick Generally, you use overloaded constructors to make a class more flexible. This is important when you're building a multipurpose class that will be used in many programs. If you're not expecting to reuse your class, multiple constructors are not necessary. However, the point of building classes is to have code reuse, so overloaded constructors are never a bad idea.

```
public Fighter(string theName)
    {
      //initialize
      snowballs = 3;
      strength = 3;
      name = theName;
    } // end constructor
```

Throwing a Snowball

The most exciting part of the game happens when a fighter throws a snowball. The throwSnow() method handles the act of throwing a snowball. The program uses a random number generator to determine whether the snowball hits its target. It requires a parameter to determine the range between the thrower and the target:

```
public bool throwSnow(int range){
      //calculates likeliness of a hit at a given range
      //returns true if snowball hit

      bool hit = false;
      int myRoll;
      Random roller = new Random();
      if (snowballs <= 0) {
        Console.WriteLine ("{0} is out of snowballs!", name);
      } else {
        myRoll = roller.Next(10);
        if (myRoll > range) {
          hit = true;
        } // end hit if
        snowballs--;
      } // end out of snowballs if
return hit;
    } // end throwSnow method
  } // end fighter
} // end namespace
```

The roller variable is an instance of the Random class. The myRoll variable holds a random value between 0 and 9.

The throwSnow() method checks whether the player has snowballs remaining. If the player has zero snowballs, the throw is canceled. If the player has snowballs remaining, the program determines whether the throw hits its target.

I used a simple algorithm to determine whether the throw succeeds. I wanted the players to trade safety for accuracy. The farther away the target is from the thrower, the less likely the target will be hit (and the less danger the thrower is in because the robot uses similar calculations to determine likelihood of a hit). To implement this algorithm, I used the Next() method of the Random object to obtain a number between 0 and 9. (If you send an int parameter to the next method, it returns a positive integer smaller than the parameter.)

If the random number is larger than the range, the snowball hits the target. As the range gets closer, it becomes easier to hit the target.

Hint Creating complex algorithms is tempting, but simple approaches are usually better. There's absolutely no scientific basis to the way I figured out the algorithm for determining hits. It was the simplest way I could think of to make a random value more likely to hit when the throwers are closer. Start simple and make things more complex only when you have a good reason to do so.

Building the Robot Fighter

The basic Fighter class is functional enough for most purposes. However, it relies on human control. To make a credible robot player, I needed a player almost like the human fighter, but with the capability to make autonomous decisions. Designing a challenging computer opponent can be difficult, but designing rudimentary computer behavior is easy. The robot fighter is inherited from the ordinary Fighter class. This means that when I built the robot class, all I needed to create were those elements of the robot fighter that add the robot player's very rudimentary intelligence. Inheritance can be magical.

Initializing the RoboFighter

The RoboFighter class is a classic candidate for inheritance. Because the class will be simply an enhancement of the Fighter class, it's natural to extend RoboFighter from Fighter. The RoboFighter will have name, snowballs, and strength properties and the throwSnow() method. All these characteristics are inherited from the Fighter class. The RoboFighter has a couple new characteristics of its own.

```
namespace Snowball
{
  /// <summary>
  /// RoboFighter
  /// A computer-controlled snowball fighter
  /// derived from fighter
  /// </summary>

  public class RoboFighter: Fighter
  {
    private Fighter player;

    public RoboFighter(Fighter thePlayer, string theName): base(theName)
    {
```

```
      player = thePlayer;
    } // end constructor
```

I added one private instance variable to the RoboFighter. I wanted to have access to the other player so that the robot fighter could access the human player's properties. The only constructor for the RoboFighter requires both a standard fighter (which is the human player) and the name of the robot.

Choosing the Robot's Play

All the key artificial intelligence comes in the choosePlay() method added to the RoboFighter. Basically, the choosePlay() method uses another random number generator to determine which play the robot should make:

```
public int choosePlay(int range){
  int thePlay;
  Random roller = new Random();
  if (snowballs <= 0){
    //make a new snowball if out of them
    Console.WriteLine(name + " is making a snowball");
    snowballs++;
  } else {
    //decide to throw or move
    thePlay = roller.Next(6);
    switch (thePlay){
      case 0:
        //go closer
        Console.WriteLine("{0} moves closer.", name);
        range--;
       break;
     case 1:
       //back up
       Console.WriteLine("{0} backs away.", name);
       range++;
       break;
     case 2:
       //make a snowball
       Console.WriteLine(name + " is making a snowball");
       snowballs++;
       break;
     default:
       //otherwise, throw a snowball
       Console.WriteLine("{0} throws a snowball", name);
       if (throwSnow(range)){
         Console.WriteLine("{0} has been hit", player.name);
         player.strength--;
        } else {
          Console.WriteLine("{0} missed you.", name);
        } // end if
        break;
    } // end switch
  } // end out of snowballs if
  return range;
} // end choosePlay
```

The roller variable holds a Random object, and thePlay is designed to hold a random value between 0 and 5.

I wanted the robot to throw a snowball about half the time. The other plays should be random

motion forward or back, or perhaps building a snowball. However, if the robot player is out of snowballs, it should immediately make one. If there are snowballs, I used the roller to generate a 0–5 integer. I used a switch statement to determine what the robot player will do. If the robot rolls a 0, it will move closer to the human. If it rolls a 1, it will back away. If it rolls a 2, it will build another snowball. Any other roll will result in a throw.

If the robot successfully hits the player, it modifies the player's strength parameter.

The easiest way to change the game is to modify the logic in this method. This very simple logic provides an interesting opponent, but one that is easy to beat. To make the program more interesting, you might want to tweak the logic. I suggest some improvements in the challenges at the end of the chapter.

Creating the Main Menu Class

The main menu of the Snowball Fight program handles the overall game logic but passes most of the details to the two fighter classes. It handles most of the actual interaction between the user and the game.

Creating Instance Variables and Main() Method

The setup of the main menu is straightforward. Instance variables in a container class like the main menu generally hold information that your entire program will need.

```
{
  /// <summary>
  /// A snowball fight against a robot opponent
  /// demonstrates object-oriented programming
  /// Andy Harris, 12/23/01
  /// </summary>
  class MainMenu
  {
    int range;                //distance between fighters
    Fighter player;           //human player
    RoboFighter opponent;     //robot opponent
    bool keepGoing = true;    //controls main loop

    static void Main(string[] args)
    {
      MainMenu mm = new MainMenu();
    }
```

The range variable holds the distance between the adversaries. This value belongs to the menu because it is common to all three classes in the program. I sent the range as a parameter to the throwSnow()and choosePlay() methods.

Creating the Menu's Constructor

The menu class constructor has two main parts. The first part sets up a few variables and initializes the two opponents, and the second part manages the interactions with the user. First, take a look at the part that initializes all the other objects:

```
public MainMenu(){
  int choice;
  string name;
```

```
  //set up the contestants
  Console.Write("What is your name? ");
  name = Console.ReadLine();
  player = new Fighter(name);
  Console.Write("What's your opponent's name? ");
  name = Console.ReadLine();
  opponent = new RoboFighter(player, name);
  range = 10;
```

Choice will hold the human player's most recent menu selection. The string variable name will be used to get names from the user for the human– and robot–controlled fighters. Because both the fighter classes require a name as part of the constructor call, I had to get name values from the user before instantiating the classes. I set the initial range to 10, which means that neither player will be able to hit the other without moving closer.

Managing the Responses

The main logic of the program consists of analyzing input from the menu. A while loop continues as long as the keepGoing variable remains true. The menu itself will be drawn in the displayMenu() method (described next.) After displaying the menu, the program uses a switch statement to determine what action the player wants to take.

```
  while(keepGoing){
    choice = displayMenu();
    switch (choice){
      case 0:
        //quit
        Console.WriteLine("quitting");
        keepGoing = false;
        break;
      case 1:
        //make a snowball
        player.snowballs++;
        break;
      case 2:
        range--;
        if (range < 0) {
          range = 0;
        } // end if
        break;
      case 3:
        range++;
        break;
      case 4:
        if (player.throwSnow(range)){
          Console.WriteLine("You hit {0}", opponent.name);
          opponent.strength--;
        } else {
          Console.WriteLine("You missed {0}", opponent.name);
        } // end if
        break;
      default:
        Console.WriteLine("you said {0}", choice);
        break;
    } // end switch
    range = opponent.choosePlay(range);
    checkWinner();
  } // end while loop
} // end constructor
```

The displayMenu() method will return back an integer indicating what kind of action the human player wants to make. I used a switch statement to respond to the various options.

Choice 0 on the menu is quit, so if the user wants to exit, I let him or her do so by setting the keepGoing value to false. The next time through the while loop, the program will end.

Choice 1 corresponds with making a snowball, so all I need to do is increment the snowball property by 1. Notice that you can use the special increment and decrement operators (such as ++, −−, and +=) on properties just as you can on more traditional variables.

Choices 2 and 3 deal with changing the range. They are very similar, except that if you move closer, the range will decrease, and if you move farther away, the range will increase. I decided to check for a lower bound when range was decremented because it doesn't make sense for the range to be less than 0. I wasn't worried about checking an upper bound because the player cannot win if he or she is too far away to hit.

Hint Whenever you increment or decrement a variable, think about whether you need to implement a boundary−checking routine. It isn't always necessary, but if you forget it and need it, your code will end up crashing at some inopportune time.

Choice 4 involves throwing the snowball. Most of the snowball−throwing logic lives inside the Fighter class, but I still needed some logic here. The throwSnow() method returns back a Boolean value of true if the snowball hits the mark. If the player hits the robot, the program responds and decrements the robot player's strength property.

These are the only choices on the menu, but users will be users—they will enter odd things into your program. The default clause should check all these strange inputs. The game design adds a built−in penalty for mistaken input because the human player forfeits the opportunity to move, throw, or make a snowball during the turn. However, the computer opponent can still make a play.

After allowing the human player to make a selection, the robot player has a turn. All the code for managing the robot player's turn is in its choosePlay() method. Notice that I passed the current range to the choosePlay() method, and it returns the new range after determining what the robot player does.

Checking for a Winner

After both players have an opportunity to slug each other with frozen slush, it is necessary to see whether somebody has won the game. This is done with a call to the custom checkWinner() method. Actually, it checks whether a player has lost and indicates which player has won. It's easy to figure out:

```
public void checkWinner(){
  if (opponent.strength <= 0){
    Console.WriteLine("You win!");
    keepGoing = false;
    Console.ReadLine();
  } else if (player.strength <= 0){
    Console.WriteLine("You have been defeated");
    keepGoing = false;
    Console.ReadLine();
  } // end if
} // end checkWinner
```

If either fighter's strength drops to 0 or below, the other fighter wins. In either case, I set keepGoing to false so that the program would exit the next time through the Main loop.

In the Real World

Sometimes you describe something in English that turns out to be very difficult to convert into programming terms. In the snowball fight, saying "Check whether somebody has won the game" in your algorithm is easy, but the actual condition is much more difficult to write because winning is not really defined in this game. Instead, a player wins only if the opponent loses. Sometimes you must modify your algorithm so that it still retains its meaning. Seeing whether anybody has won is difficult, but finding out whether somebody has lost is easy, and the meaning is the same.

Displaying the Menu

The code for displaying the main menu is straightforward. The main menu consists of two parts: the scoreboard describing various statistics and the list of choices facing the human player. Both are accomplished through a set of Console.WriteLine() statements, as shown in the following code:

```
public int displayMenu(){
    int choice;
    Console.WriteLine("Stats:");
    Console.WriteLine("  distance: \t{0}", range);
    Console.WriteLine("  snowballs: \t{0}", player.snowballs);
    Console.WriteLine("  {0}: \t{1}", player.name, player.strength);
    Console.WriteLine("  {0}: \t{1}", opponent.name, opponent.strength);
    Console.WriteLine();
    Console.WriteLine("0.  Quit");
    Console.WriteLine("1. Make a snowball");
    Console.WriteLine("2. Move closer");
    Console.WriteLine("3. Run away");
    Console.WriteLine("4. Throw");
    Console.WriteLine();

   Console.Write("Indicate your choice: ");
   choice = Convert.ToInt32(Console.ReadLine());
   Console.WriteLine();
   Console.WriteLine();
   return choice;
 } // end dispayMenu
```

I needed to change the string value that comes from a Console.ReadLine() call to the integer value that displayMenu has to send out.

Summary

This chapter has led you through some of the most important ideas in modern computer programming. You have learned how to customize the creation of your classes through constructors. You have learned how to overload constructors by adding multiple constructors with different parameter sets. You have used inheritance to create new types of classes that inherit characteristics of existing classes. You have learned how polymorphism can be used to customize methods for new classes. All these new skills together give you the ability to create powerful new objects and manipulate them in interesting ways.

Challenges

- Create your own new forms of the Critter class, inherited from the original critter.
- Add a trick method to your critter so that it can do various types of tricks. Perhaps the happier a critter is, the better trick it can perform.
- Make a two–player version of the snowball fight. You won't need the RoboFighter class, but you will need two copies of the Fighter class. You'll need to modify the main menu code a little, but two player games (at least, on the same computer) are easier to write than artificial intelligence.
- Improve the performance of the robot fighter. Perhaps have it throw snowballs only when it is closer than five units away, or have it always run away after being hit so it can build more snowballs.
- Make a tournament. Allow the user to choose human and computer players. Then build a tournament that pairs various fighters together, ultimately choosing a champion. Hint: You can reuse the existing program almost in its entirety. The main difference will be adding another program outside of it to control the tournament. Each individual snowball fight will invoke the Snowball Fight program.

Chapter 6: Creating a Windows Program: The Visual Critter

Overview

The first programs in this book are much like the programs in the early high–level languages, such as COBOL and FORTRAN. These programs work primarily on the command line and begin with straight–line logic. As you have progressed in your own programming, you have used more complex logical structures, such as loops and branching statements, and you have played around with more complex data structures, such as variables. The history of programming follows the same progression. In the past two chapters, you looked at object–oriented programming, which represents the current thinking in program design, but your programs still do not seem very modern because they are written on the ugly and unfriendly console. In the next few chapters, you will look at how C# is used in Windows programming. The programs you will write feature the capabilities of the Windows operating system, with all the graphical features you have come to expect from such a system.

Now you will explore how to build a program with a Windows interface. In this chapter you will

- Learn the basic concepts of a graphical user interface (GUI).
- Use the IDE to build an interface.
- Navigate the System.Drawing and System.Windows namespaces.
- Add the most common GUI elements (text boxes, labels, images, scroll bars, and multiple selection elements) to your forms.
- Write event–handling code.
- Use event–driven programming.

Introducing the Visual Critter

As the final exercise for this chapter, you will build a critter program that has a graphical interface. This program will look and feel more like the programs you encounter on the Windows operating system. As usual, an image or two of the program is more helpful than just a description. Figure 6.1 demonstrates the Visual Critter in its default state.

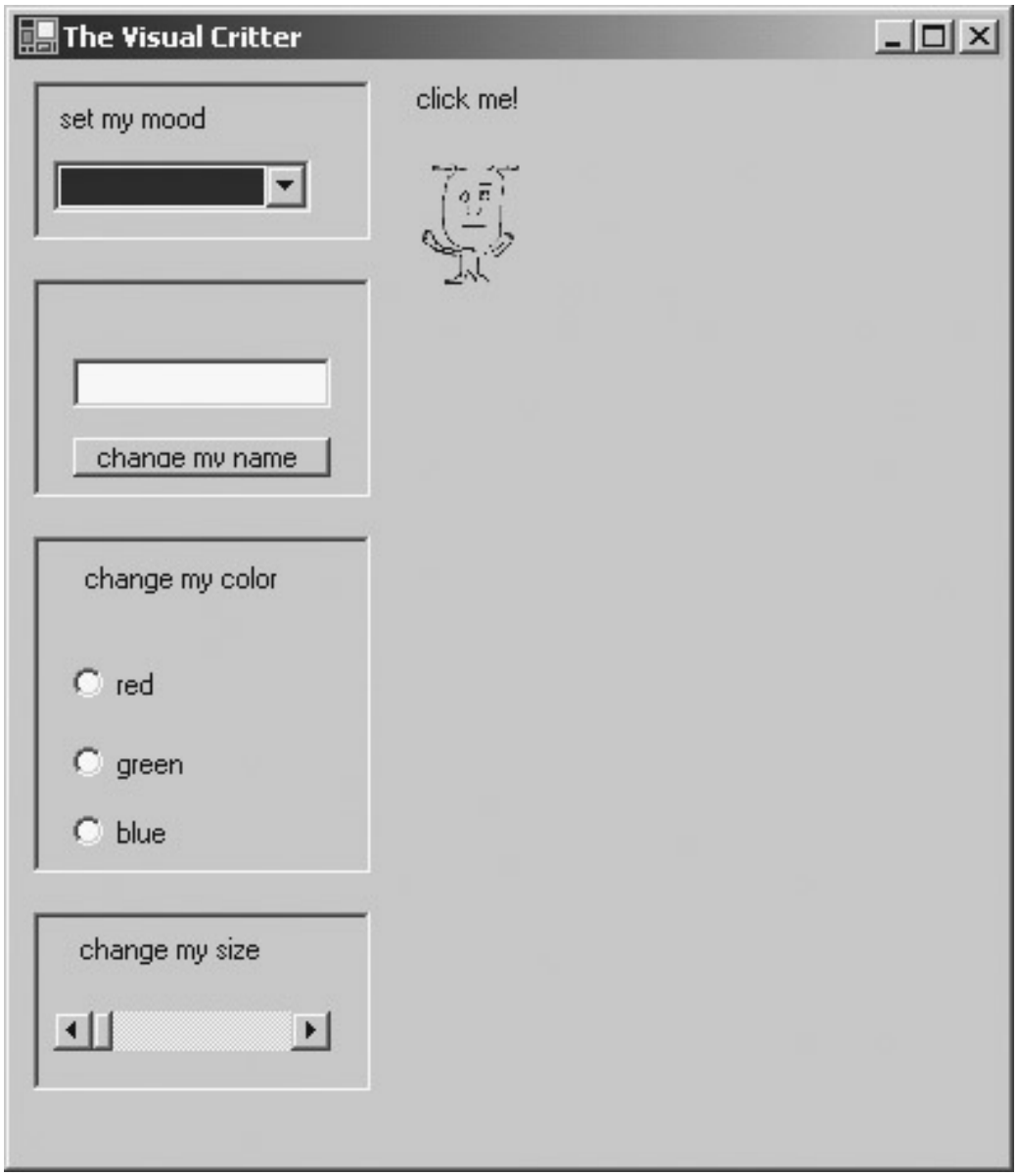


Figure 6.1: This critter has a picture and some Windows–style controls.
You can see from Figure 6.1 that the newest version of the critter has much more visual appeal than the older versions. Modern users have become used to the convenience of a graphical interface, with all the various components such as text boxes, scroll bars, and command buttons. This version of the Visual Critter features several standard graphical controls.

Before diving into how all these controls work, you should play around with the program and see what it does. This is a highly interactive program, so you should open it from the CD–ROM and look at it live. I'll show you a few of the highlights now in case you aren't near a computer.

Figure 6.2 demonstrates what happens when you click the critter image.

Figure 6.2: When the user clicks the image, the critter speaks.

I have attached code to the image so that when it is clicked, a message appears. The critter asks the user to change the critter's name. Take a look at Figures 6.3 and 6.4 to see how the user can edit the name in the rectangle on the left side of the form.

Figure 6.3: The user can rename the critter by typing in the little box.

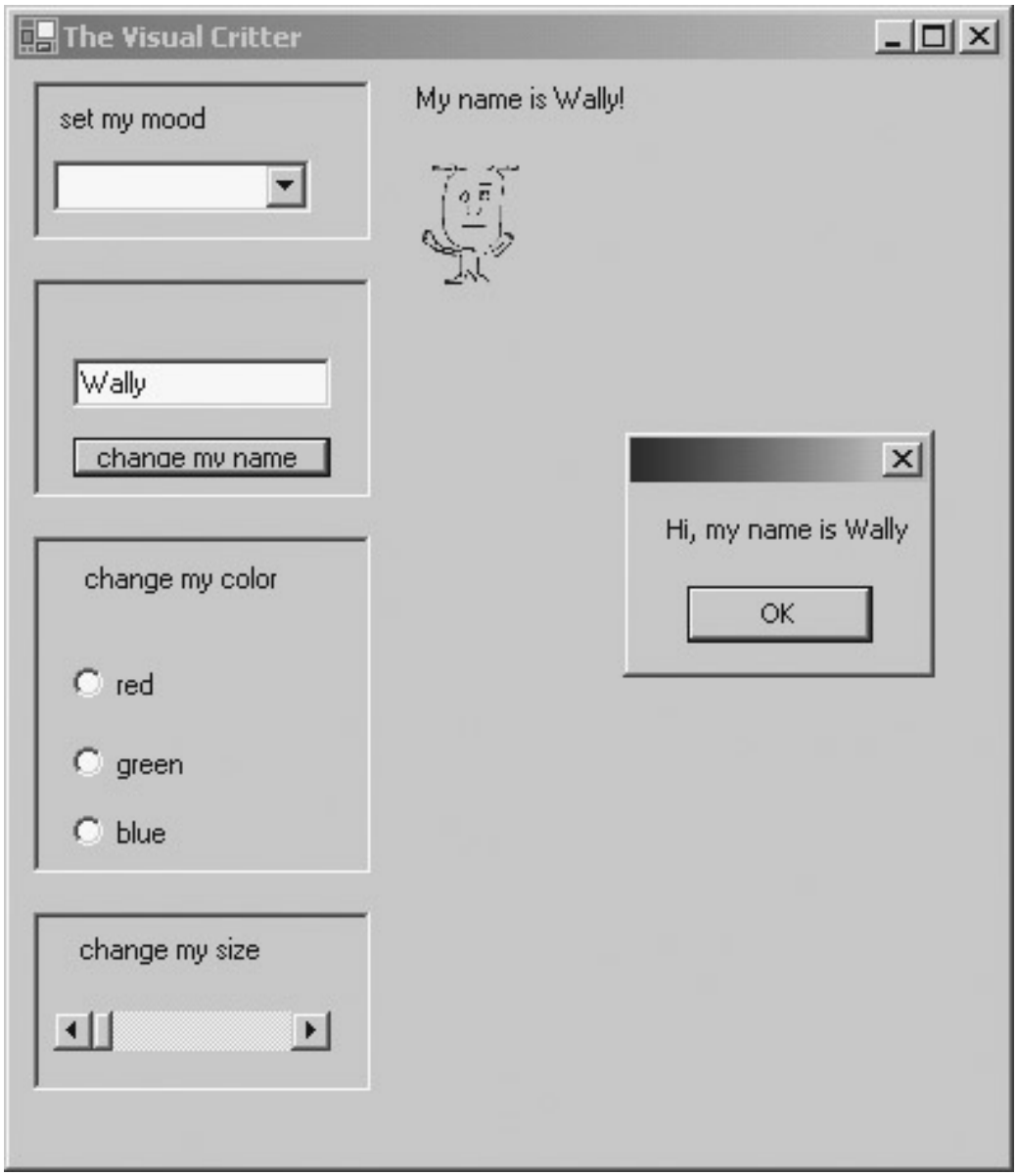


Figure 6.4: After the user clicks the Change My Name button, the critter reports its new name. Any subsequent clicks on the critter will return the new name.

You can do other interesting things with the critter. You can change its mood by clicking the drop–down list of various temperaments. Figure 6.5 illustrates the critter after the user chooses a different mood.

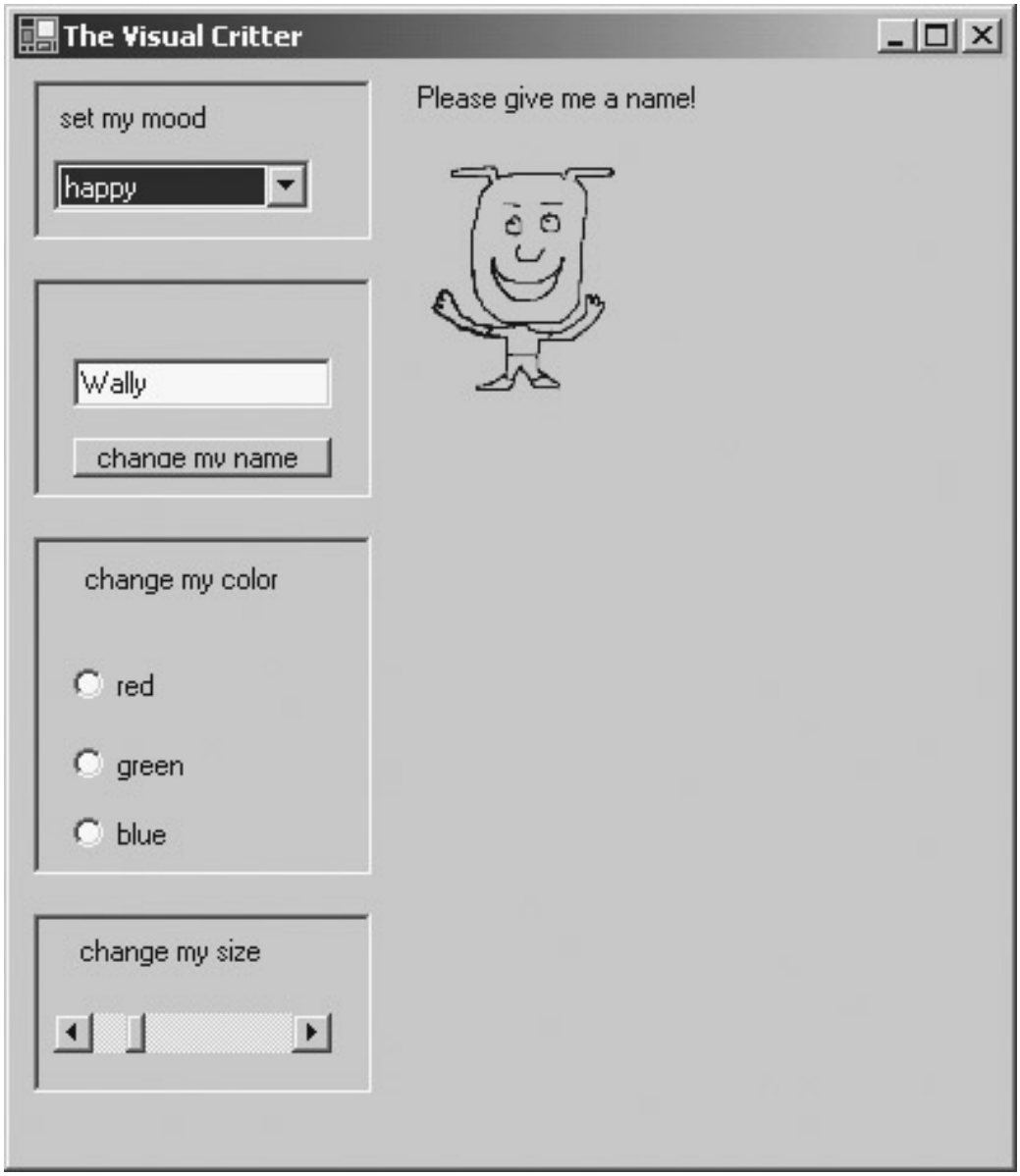


Figure 6.5: When the user chooses a new mood from the drop–down list, the image changes accordingly.

The user can also change the critter's size, by manipulating the scroll bar. Changing the location of the rectangle inside the elevator shaft changes the size of the critter image. Figures 6.6 and 6.7 illustrate this phenomenon.

Figure 6.6: The elevator shaft is near the top, and the critter image is very small.

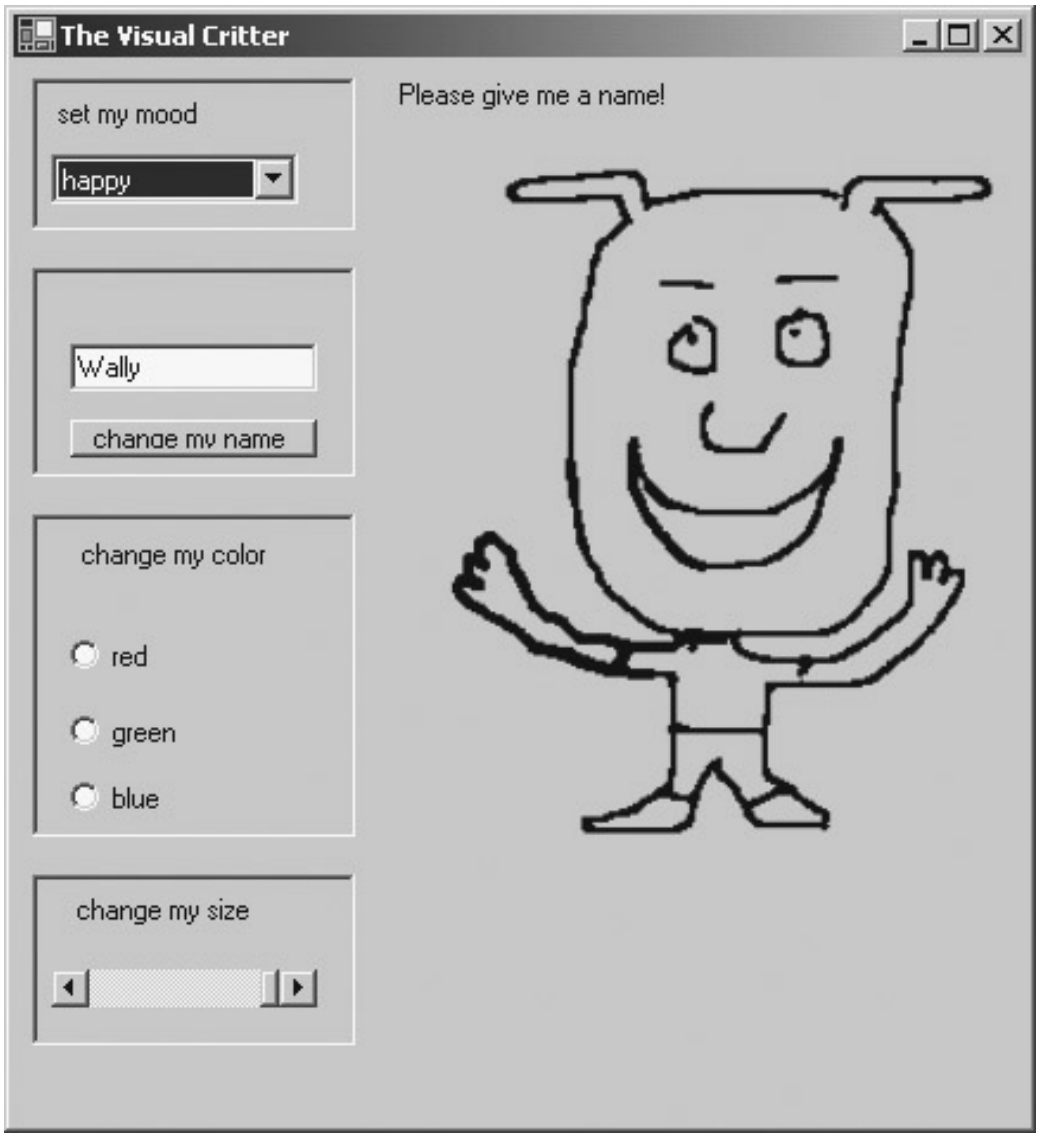


Figure 6.7: When the user moves the scroll bar down, the critter grows.
You can also change the color of the background on which the critter resides. Illustrating these color changes on a black–and–white page is impossible, so you need to experiment with the program to see this phenomenon.

Creating a Windows–Style Program with a GUI

The Windows operating system, like most modern computer operating systems, provides a *graphical user interface (GUI)*. The GUI, pronounced “gooey,” (I love it when technical terms sound silly) has several purposes. It is intended to make the user’s life easier. Manipulating a mouse and clicking screen elements is presumably easier than typing commands from a text–based menu. Visual elements on the screen are more aesthetically interesting than the kinds of programs you have seen so far in this book. Graphical elements, such as text boxes, list boxes, and scroll bars, also have practical benefits. Many of them are designed to limit user input. This prevents the user from entering invalid data and minimizes the error–checking code the programmer must write. Also, most graphical elements are already familiar to computer users. If you use only standard components in your programs, most of your users will already know the basic operation of these objects. Finally, the visual components are predefined by the Windows operating system, so you don’t have to write all the code for them every time. All the visual elements are available as objects.

You can simply create an instance of the button object, for example, and you have a button. Most of the special visual objects you use to create Windows programs are housed in the System.Windows.Forms namespace.

Thinking Like a GUI Programmer

Programs that feature a graphical user interface are different from console programs. The following sections explain these differences.

Objects Are Everywhere

The explosion in object–oriented programming happened at the same time that graphical user interfaces entered the mainstream. This is no coincidence. Graphical interfaces lend themselves beautifully to the object–oriented paradigm. The window itself is an object, as are all the various buttons and widgets on the screen.

The User Is in Control

In a console–based application, the programmer is king. You determine everything, and the program flows according to your whims. In short, the user can only respond to the program. In the graphical universe, the user is the king. Your program is set up as a playground for the user, and the programmer's job is to anticipate the user's actions. This makes GUI programs friendly (usually) for the user but more challenging for the programmer to write.

Programs Respond to Events

The entire design of a GUI program is different from one based on a console. Console–based programs generally have well–established beginnings, middle segments, and ending places, with one main loop. A GUI program is a much more fluid affair. You generally do not write a main loop at all! Instead, the nature of a Windows program implies a main loop that constantly repeats until the program is ended. This main loop patiently waits for the user to do something. The user's action (such as pressing a button or moving the mouse) triggers an *event*. Most of the code in a GUI program comprises responses to events.

Problems Are Solved Differently

In the console world, you have to build everything, and your program follows a logical flow. You can design a flowchart and expect the code to follow a similar structure. In a GUI program, the problem–solving process is more visual. You often begin by sketching out a screen. Then you figure out which controls and objects will bring that screen to life. Most of your code has to do with choosing objects, manipulating the properties of those objects, and then responding to events that occur when the user interacts with these objects.

Events Are Added to the Object Model

GUI programming requires that you expand your thinking about objects. You are already accustomed to objects' having properties and methods, but user interface objects (as well as other kinds of objects) also require a special characteristic called an *event*. If a property is an adjective (it describes a characteristic of an object) and a method is a verb (it describes what an object can do), an event is an interjection, such as "Ouch!" or "Help!" Events are messages sent to other objects in the system. Normally, you use an event to indicate that something has happened—the user has clicked a button or moved a scroll bar, for example. Other events can be automatically triggered by

the operating system or the passage of time.

Creating a Graphical User Interface (GUI)

To learn the basic concepts of designing GUI forms, you'll re–create the classic Hello World program from Chapter 1, "Basic Input and Output: A Mini Adventure." This time, you will write it as a Windows GUI program. Although creating a graphical interface is more challenging than designing a console application, the Visual Studio IDE provides features that greatly simplify the process.

Creating a Windows Project

To begin, you create a C# program as usual. When you are asked for the type of project, choose Windows Form from the templates list box (see Figure 6.8).

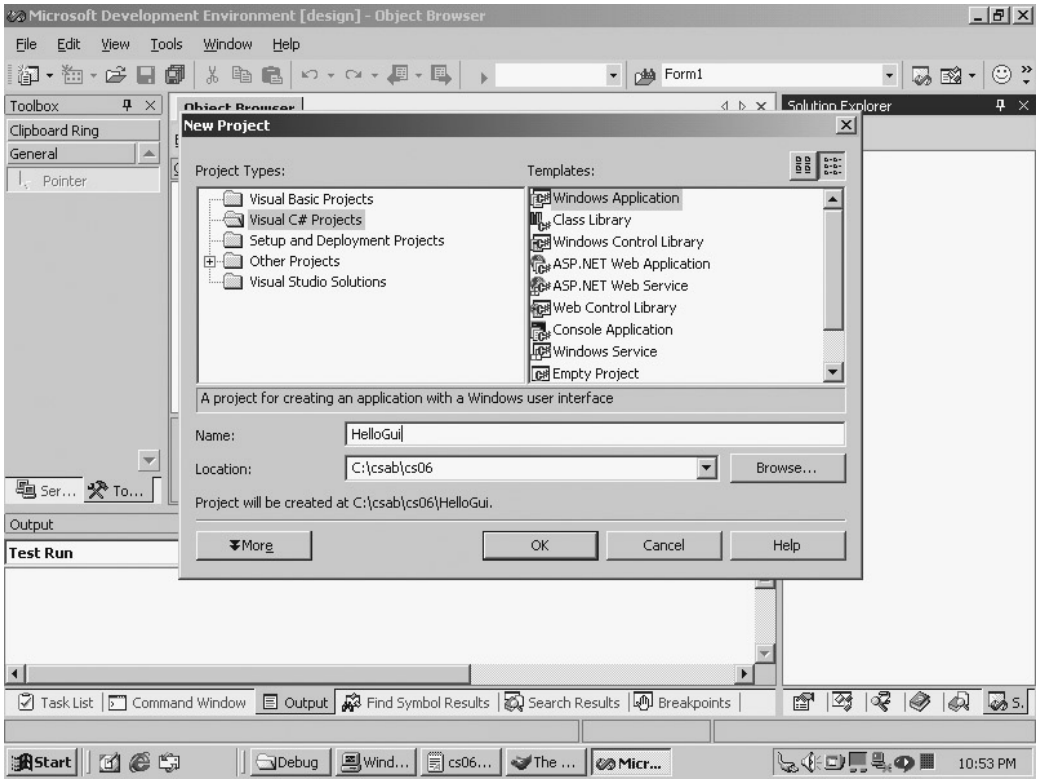


Figure 6.8: You can choose to make a Windows project when you start a new C# project.

The editor will continue to load as usual, but the screen will be a little different from what you are used to.

Trap Be sure to choose a good name when you are creating the project because it is very difficult to change later. Leaving the name "WindowsFormApplication1" then changing it to "GlitterCritterLiveInVegas" requires several painful steps that can cause the program to not compile.

Using the Form Builder

The IDE has a new feature when you are building a Windows–style project. Figure 6.9 shows a graphical editor with a form in place. The visual form designer contains a number of panels and features to simplify the creation of GUI interfaces:

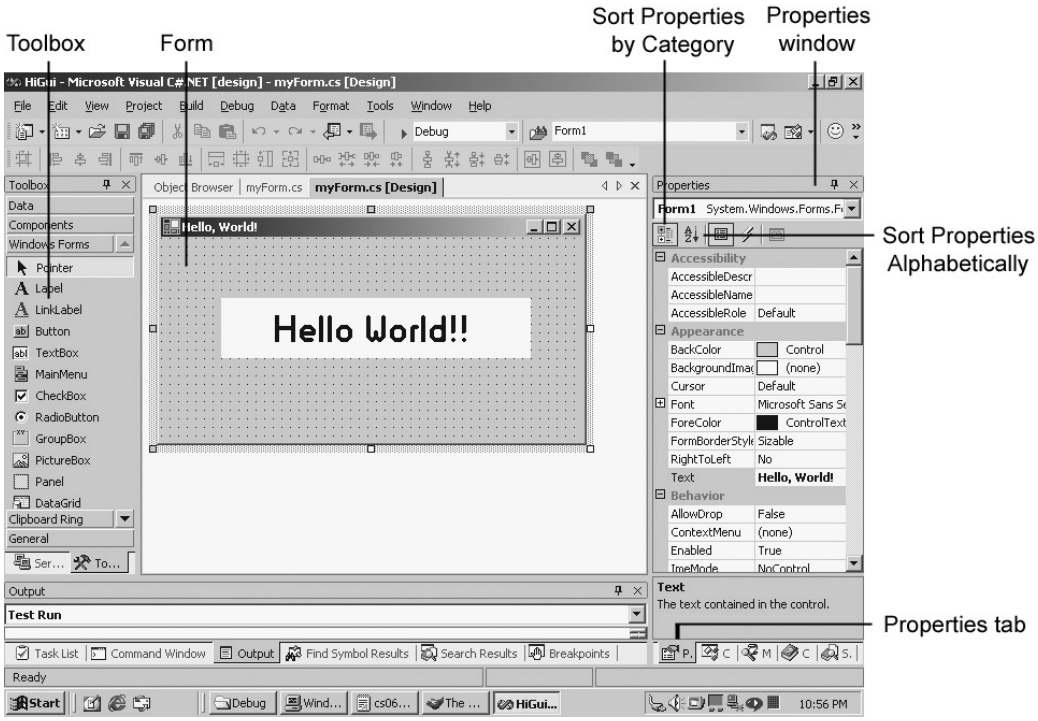


Figure 6.9: When you build a Windows project, the IDE changes to add graphical features.

- **Toolbox.** Features components you can place on your form. You can highlight any object and draw it onto the form, as in an image–editing program.
- **Form.** Another page in the main area where the code, object browser, and help screens have appeared before. You can resize the form by dragging on the small squares around its edges.
- **Properties window.** Displays all the properties of the currently selected object. In this case, it shows all the properties of the form object that makes up the program. You can dynamically change a visual object's properties by manipulating the Properties window.
- **Properties tab.** Used to select the Properties window from the many windows available in this pane.
- **Sort Properties Alphabetically.** Alphabetizes all the properties in the Properties window by property name. This can be useful when you know the exact name of the property you want to change.
- **Sort Properties by Category.** Displays the properties by category. For example, all the properties dealing with the object's appearance are grouped together, as well as the properties dealing with the object's behavior. I have found the categories to be unintuitive (what's the difference between appearance, design, and layout properties, for example?) However, when you become familiar with the way properties are organized, this is an easy method to find what you need.

Trap Sometimes, one of the windows (especially the Properties window) "disappears" while you are working. The highly flexible design of the .NET IDE can make the exact locations of various elements hard to predict. If you need access to a window and cannot see it on the screen, you can always select it from the View menu. In particular, when the Properties window is missing, you can find it by clicking the tabs in the right column of the editor.

Changing the Form's Properties

If the form is *highlighted* (the sizing squares around its perimeter are visible), you can change its properties in the Properties window. Examine the form's title bar. When you begin, the title bar reads Form 1. This becomes the title bar of your finished program. The title bar usually contains the

name of the program and sometimes other information. In any case, the form's title bar is related to the form's text property. Most visual objects have a text property, which allows you to change text on the object. Find the text property in the Properties window, and type **Hello World!!** in the box.

Trick The Properties window is a very important part of visual programming, so here are a few tips for using it. I like to make it larger than the default size. You can resize it by dragging its borders. You can also scroll up and down in the window if there are more properties than you can see at one time on the screen. You can display the properties alphabetically by choosing the button that has "AZ" on it at the top of the Properties window, or you can sort by category, by choosing the icon immediately to the left of the "AZ" icon.

For now, simply change the form's text property. If you like, you can play around with other properties, too. Try to change the background color of the form. Experiment!

Trap Because of a flaw in the IDE design, the program will not run properly if you change the name of your startup form. In the next section you will learn how to fix this problem, but for now, just leave the form's name as *Form1*.

Adding a Label to the Form

Although you have changed the form's text to Hello World!!, the greeting is too subtle for most users to notice. Larger text centered in the middle of the screen would be better. This is easy to do. Look at the Toolbox, which should be on the left side of the screen. If it is not visible, select it from the View menu. The Toolbox contains visual objects you can place on your forms. These special objects are sometimes called *controls* or *components*. You select the component by clicking it in the Toolbox and then draw the component onto your form. It works much like a painting program, except that rather than paint colors on a palette, you paint controls onto a form.

Hint Like the other windows in the .NET IDE, the Toolbox has several faces. If you don't see the controls you are looking for, make sure that you have selected the Windows Forms tab in the Toolbox. Also, the number of controls available can exceed the space allocated for the Toolbox. You use the up and down arrows on the Toolbox window to scroll through the controls available to you.

After a control is placed on the form, you move it by dragging it around. You resize it by manipulating the sizing rectangles on the sides and corners of the control.

Changing the Label's Properties

Like any other controls, the label supports intriguing properties. The label's text property determines what text will be placed in the label. Figure 6.10 shows the changes I made to the label—see whether you can match my changes. I modified the font name, font size, TextAlign, and BackColor properties to get the effect I was looking for. Of course, you can modify the program however you like.

Figure 6.10: The program says a Windows–style hi, without a line of code!

Running the Program

The .NET editor makes creating a simple GUI program very easy. When you click the Run icon, you see the working version of your program. Figure 6.10 features my version.

This program looks good and is very easy to write. You don't add a single line of code. However, the program *is* based on code, so you need to understand how the code is built behind the scenes. Sometimes you will want to write code without the graphical editor You will learn how to do this in Chapter 8 "Arrays: The Soccer Game." More often, you will find that the editor makes a mistake you must correct. Of course usually you will add more code so that your program does something. Looking at the code generated by the editor makes sense.

Examining the Code of a Windows Program

When you create a new Windows application, the editor shows you a graphical *representation* of the form, not the form itself. This graphical window is called the *Visual Designer*. Your program is still based on code. The Designer view simply gives you an alternative way of viewing and modifying the code. To see the underlying code, choose Code Window from the View menu. Figure 6.11 shows the editor displaying the code for the Hello GUI.

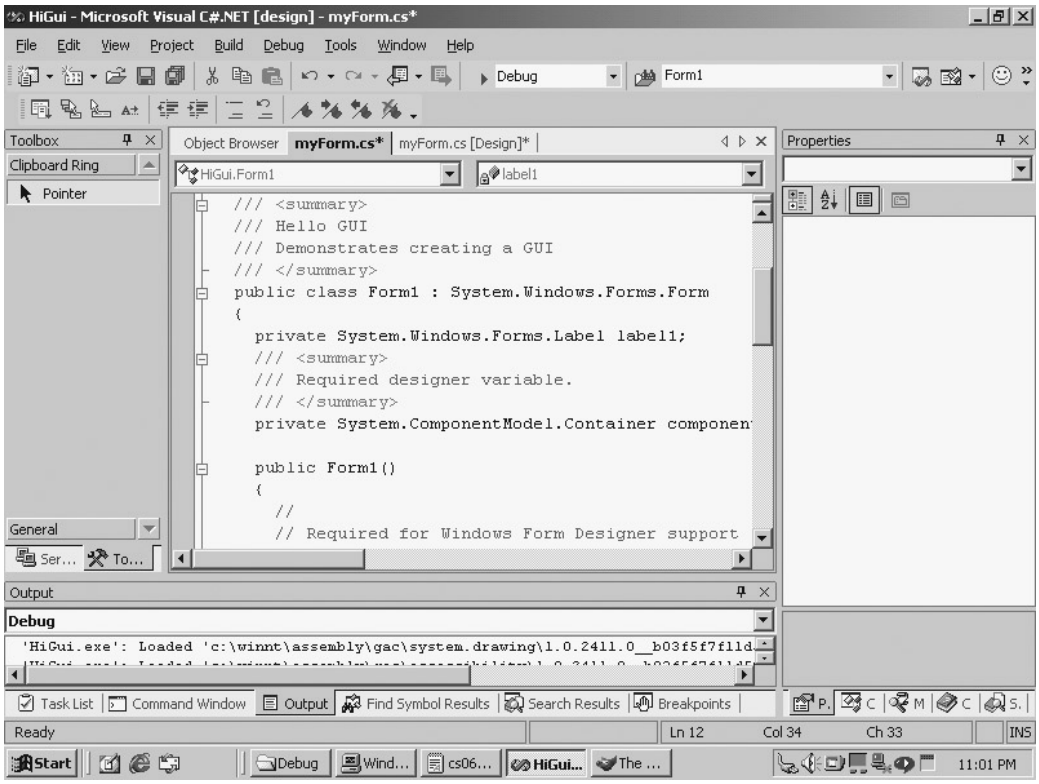


Figure 6.11: This code creates the Hello GUI program.

As you scroll through the code, you see that the editor wrote a lot of code. Although the editor writes this automatically, it is not perfect. I'll show you the automatically generated code and explain what it is doing.

Adding New Namespaces

When you create a console program, the editor presumes that you will want access to the System namespace. In a Windows program, you need access to other namespaces as well. All the visual components are objects, and most of them live in the System.Windows.Forms namespace. Additionally, all the objects used to position controls on a form belong in another important namespace, System.Drawing. When you create a Windows program, the C# editor automatically attaches references to these two critical namespaces, and a few others as well:

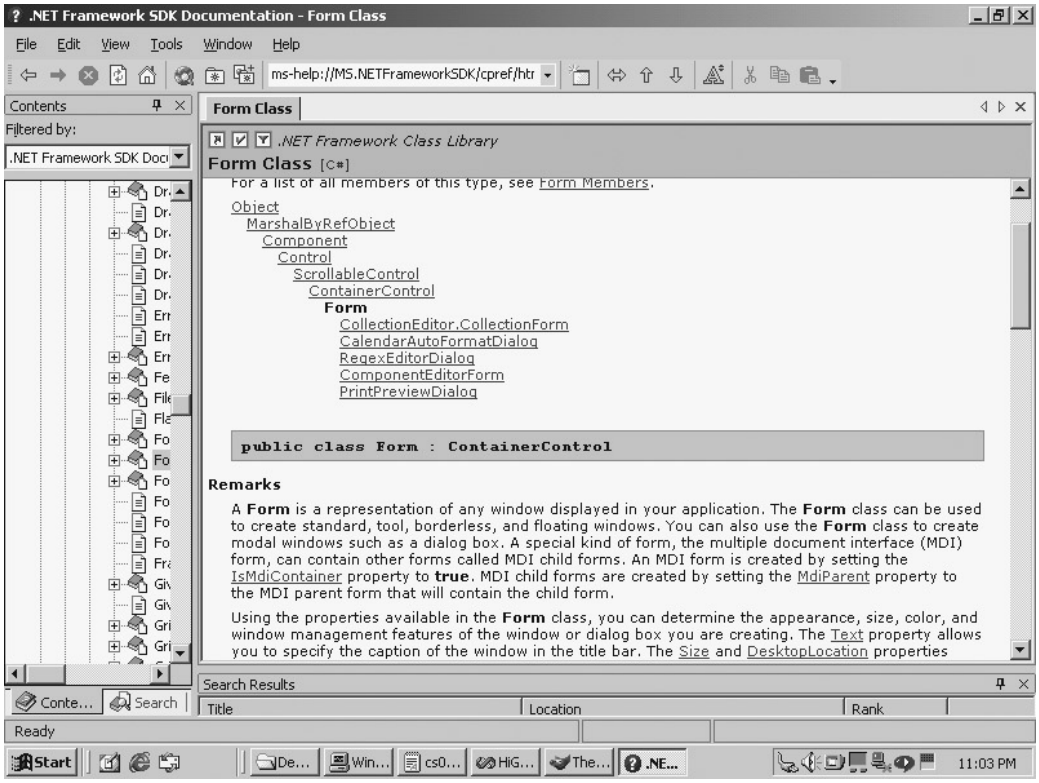
```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;
```

For this simple program, you need not worry about the other namespaces, but it doesn't hurt to leave them in.

The System.Windows.Forms Namespace

Most of the controls visible on the toolbar are housed in an important namespace, System.Windows.Forms. If you include access to this namespace, your programs can use forms and all the other controls on the Toolbox. Not surprisingly, nearly every Windows–based program uses this namespace. Peruse the object browser or documentation to see all that this namespace has to offer. Be warned, though. The namespace is *huge*, and you don't need to memorize all the

various elements. It's just nice to know what's there. Each of the components is a full–blown object, with properties, methods, and events. To demonstrate some features of the namespace, Figure 6.12 shows the documentation for the Form class.


.NET Framework SDK Documentation - Form Class
File Edit View Tools Window Help
Contents
Filtered by:
Form Class
.NET Framework Class Library
Form Class [C#]
For a list of all members of this type, see Form Members.
Object
MarshalByRefObject
Component
Control
ScrollableControl
ContainerControl
Form
CollectionEditor.CollectionForm
CalendarAutoFormatDialog
RegexEditorDialog
ComponentEditorForm
PrintPreviewDialog
public class Form : ContainerControl
Remarks
A Form is a representation of any window displayed in your application. The Form class can be used to create standard, tool, borderless, and floating windows. You can also use the Form class to create modal windows such as a dialog box. A special kind of form, the multiple document interface (MDI) form, can contain other forms called MDI child forms. An MDI form is created by setting the IsMdiContainer property to true. MDI child forms are created by setting the MdiParent property to the MDI parent form that will contain the child form.
Using the properties available in the Form class, you can determine the appearance, size, color, and window management features of the window or dialog box you are creating. The Text property allows you to specify the caption of the window in the title bar. The Size and DesktopLocation properties
Search Results
Title
Location
Rank
Ready
Start
11:03 PM
Figure 6.12: The Form class is a very powerful part of the System. Windows.Forms namespace, with many useful characteristics.

In the Real World

Even the pros don't memorize all these details; they usually have a help screen and keep a reference book around.

You need to know how to dig around and find out the characteristics of components because learning them all is impossible. The more important skill is knowing how to find what you need when you need it. You solve most of your GUI problems by thinking about what kind of component might help. If a control or other object isn't the solution, a property or method of an object probably is. As you can see, understanding how objects work is the key to modern programming.

Inheritance and Related Components

When you look at the documentation for the Form class, you see that the form object supports a huge number of properties and methods (as well as events, which you'll investigate shortly). However, the form object is even more powerful than it appears! If you look back at Figure 6.12, you can see a miniature outline showing how the form is related to some other objects. This shows the form object's family tree. Because the form is a descendant of the ContainerControl class, it has features that enable it to hold other controls. This class is descended from the ScrollableControl class, which is inherited from Control.

Each ancestor of the Form class adds new capabilities to the object. This means that the Form can

act just like other controls but also has new features of its own. This is good for you. When you learn which features a Control has, for example, you have a head start on everything that derives from the control object. For example, the Control class has a backColor property that determines the object's background color. Because almost all the widgets you can place on a form (and the form itself) have the Control class in their family tree, all of them have access to the BackColor property.

The authors of the .Net framework cleverly used inheritance to simplify their work. (Adding a BackColor property to every component is unnecessary because most components are derived from the Control class, which already has the property.) It also makes your job as a programmer easier. When you understand the inheritance pattern of an object, you know a lot about what it can do. You also have some understanding of unfamiliar components because they usually share characteristics with controls you already know.

The System.Drawing Namespace

Adding a reference to the System.Drawing namespace is necessary because this namespace provides access to certain drawing elements used in the visual designer. Most important of these is the Point class, which is used to determine the size and placement of objects on the screen. The System.Drawing namespace also defines Size and Color. You can look through this namespace in the documentation as well, but most of the classes you find there will not be useful until you have learned how to add event–handling capabilities to your programs.

The Other Namespaces

The editor provides references to other namespaces as well. They are used with enough frequency that the editor puts them in for you. For now, you can safely ignore them, but you will get a chance to play around with some of them later on.

Creating the Form Object

The editor creates all the code necessary to build a basic version of your form. Here is the code that handles basic form creation:

```
namespace HiGui
{
  /// <summary>
  /// Summary description for Form1.
  /// </summary>

  public class Form1 :
System.Windows.Forms.Form
  {
    private System.Windows.Forms.Label label1;
    /// <summary>
    /// Required designer variable.
    /// </summary>
    private System.ComponentModel.Container
components = null;

    public Form1()
    {
      //
      // Required for Windows Form
Designer support
      //
```

```
      InitializeComponent();

      //
      // TODO: Add any constructor code
after InitializeComponent call
      //
    }
```

I left this code exactly as it was created in the editor. Of course, you usually modify the code. The program starts with a summary block, where you can add documentation to your form. The code then proceeds with a definition of the Form1 object. The Form1 class is inherited from the System.Windows.Forms.Form class.

The form is a class, and it has a constructor, like any other class. In this case, the constructor is extremely simple. It calls one method, InitializeComponent(). This method is required for all code created with the Designer, and it must be called from your constructor. I'll show you the code in that method shortly. If you want to add any other constructor code, you add it after the line marked with the TODO comment. If you wish, you can take out the TODO comments altogether, as they are simply a placeholder telling you where to write your code.

Creating a Destructor

The code editor also provides an interesting method named Dispose(). Here is the code for the Dispose() method:

```
/// <summary>
/// Clean up any resources being used.
/// </summary>
protected override void Dispose( bool disposing )
{
  if( disposing )
  {
    if (components != null)
    {
      components.Dispose();
    }
  }
  base.Dispose( disposing );
}
```

The Dispose() method is automatically called whenever the Form1 class is ready to close (usually at the end of the program).

In the Real World

Dispose() is an example of a *destructor* method, which is automatically called when a class is no longer needed. The code provided here ensures that the Form class will remove itself from memory when it is no longer needed. In older languages, such as C and C++, it is very important to supply destructors such as the Dispose() method so that your program does not leave pieces of itself in the computer's memory after it closes. You might notice that after your computer has run for a long time without rebooting, it appears to be more sluggish. Programs that do not clean up after themselves properly are possible culprits. Fortunately, C# provides *automatic garbage collection,* which automates the process of cleaning up memory. At this stage of your programming career, it's safe to presume that the automatic garbage collection routines will work properly. Simply leave the Dispose() method alone, and move to the parts of the program that need your attention.

Creating the Components

The components are created and added to the form in the InitializeComponent() method (which, like everything else in this section, was automatically created for you by the editor). This method translates the components drawn on the form with the Designer to actual code that will produce the desired results. Take a look at what happens inside the InitializeComponent() method:

```
#region Windows Form Designer
generated code
/// <summary>
/// Required method for
Designer support - do not modify
/// the contents of this method
with the code editor.
/// </summary>
private void InitializeComponent()
{
  this.label1 =
new System.Windows.Forms.Label();
  this.SuspendLayout();
  //
  // label1
  //
  this.label1.BackColor =
System.Drawing.Color.White;
  this.label1.Font =
new System.Drawing.Font(
     "Glass Gauge", 27.75F,
     System.Drawing.FontStyle.Regular,
     System.Drawing.GraphicsUnit.Point,
     ((System.Byte)(0)));
  this.label1.Location =
new System.Drawing.Point(56, 56);
  this.label1.Name = "label1";
  this.label1.Size =
new System.Drawing.Size(288, 56);
  this.label1.TabIndex = 0;
  this.label1.Text = "Hello World!!";
  this.label1.TextAlign =
System.Drawing.ContentAlignment.MiddleCenter;
  //
  // myForm
  //
  this.AutoScaleBaseSize =
new System.Drawing.Size(5, 13);
  this.ClientSize =
new System.Drawing.Size(392, 189);
  this.Controls.AddRange(new
System.Windows.Forms.Control[] {

this.label1});
  this.Name = "myForm";
  this.Text = "Hello, World!";
  this.ResumeLayout(false);

}
#endregion
```

This method creates the label object (and any controls you add to a form) and sets the properties of all controls and the form. (Note that label1 ends with a numeral 1, not two *l*s.) There are a couple things to notice about the method. First, the SuspendLayout() and ResumeLayout() methods are used to suspend drawing until all the objects are configured and then to draw them all at once. Second, the comments tell you that this method was generated by the Designer and that you should not modify it by hand. This is generally good advice. You should think twice about modifying code in the InitializeComponent() method. If you do modify the code, the Designer might not recognize your changes, and the program will no longer function correctly.

Setting Component Properties

Components are objects, and like all objects, they have properties. The visual designer makes it very easy to set up an object and its properties. Because many component properties are visual in nature, you often can see the results of your property manipulation as you are editing the object, before your program even runs. As an example, here is the part of the method that sets up label1:

```
this.label1 =
new System.Windows.Forms.Label();
      this.SuspendLayout();
      //
      // label1
      //
      this.label1.BackColor =
System.Drawing.Color.White;
      this.label1.Font = new System.Drawing.Font(
        "Glass Gauge", 27.75F,
        System.Drawing.FontStyle.Regular,
        System.Drawing.GraphicsUnit.Point,
        ((System.Byte)(0)));
      this.label1.Location =
new System.Drawing.Point(56, 56);
      this.label1.Name = "label1";
      this.label1.Size =
new System.Drawing.Size(288, 56);
      this.label1.TabIndex = 0;
      this.label1.Text = "Hello World!!";
      this.label1.TextAlign =
System.Drawing.ContentAlignment.MiddleCenter;
      //
```

After the method creates a new instance of the Label class, label1., it sets properties of the label. Essentially, the program looks at the label on the form and the label's properties and sets all the properties of the label1 class so that it matches the label on the Designer. Some of the properties are designed to hold special kinds of data. For example, label1.Size requires a value of type System.Drawing.Size. Many of the property values are related to the System.Drawing namespace. For example, you can specify the color white as System.Drawing.Color.White. Most of the time, you don't have to worry about the specifics, but sometimes you need to know what kind of information goes into a property. Looking at the code generated by the Designer can be a good clue.

Setting Up the Form

The form itself is a component and is set up much like the label:

```
//
// myForm
//
this.AutoScaleBaseSize = new System.Drawing.Size(5, 13);
```

```
this.ClientSize = new System.Drawing.Size(392, 189);
this.Controls.AddRange(new System.Windows.Forms.Control[] {
this.label1});
this.Name = "myForm";
this.Text = "Hello, World!";
this.ResumeLayout(false);
```

The AutoScaleBaseSize property is used to set up how the form should be automatically scaled, based on the current font if the user decides to change the size of the form. You won't have to create it yourself. The ClientSize property determines how much room the form will have for controls. The Name and Text properties are straightforward.

The line this.Controls.AddRange(...) sets up a place in memory to hold all the controls that will be on the form. At this point, only one control is on the form, the label. If the form were more complex, each of the controls on the form would show up in a list inside this command. The AddRange() method allows you to add several controls to the form. Adding a control to a form is actually a two–step process. You designate the size, position, and other properties of the control. Then you use the AddRange() method (or another control–adding method) to create the logical link between the form and the control. Both functions are handled automatically by the Designer. (Again, I'm just showing you what is happening under the hood.)

Writing the Main() Method

Because this program is meant to stand alone, it must have a Main() method. Throughout this example, the Designer has automatically generated all the code. The Main() method is no different. Recall that the Main() method usually does nothing more than instantiate an object. The Main() method created by the Designer does the same thing but in a slightly different manner than you have seen.

```
/// <summary>
/// The main entry point for the application.
/// </summary>
[STAThread]
static void Main()
{
  Application.Run(new Form1());
}
```

The [STAThread] directive defines the default threading model for the application. Threaded programs determine how your programs will behave when other programs are running in the same system. For now, leave the STAThread line alone. This setting is fine for your current needs.

The Main() method has only one line in it and simply instantiates the Form1 object. You can use the Run method of the Application object to run any class in your namespace that has a Main() method. This works much the same as the technique you learned in the preceding chapter (creating a local variable and instantiating the class to that variable). However, because the editor created this code, it's best to leave this alone.

Trap If you change the name of the default form, be sure to check that the Application.Run() call in the Main() method points to the new form name. The Run() method does not automatically change, and the program will not run. Change the code to reflect your new form name, and you'll have no problems. This is an example of why you need to know what the Designer is doing—it isn't perfect.

Creating an Interactive Program

The editor does a good job of building simple GUI programs. However, a program isn't interesting if the user can't interact with it. The most interesting part of GUI programs is writing the code that responds to events. As usual, I'll illustrate with a simple program.

Responding to a Simple Event

The silly program illustrated in Figures 6.13 and 6.14 illustrates how to add basic interactivity to your code. Again, most of the code is automatically generated for you, but you should know what is happening behind the scenes.

Figure 6.13: The temptation is almost irresistible.

Figure 6.14: When the user (inevitably) clicks the button, the program responds to the event.

Most of the interaction between the user and a GUI program happens as a result of some kind of event. The most common type of event you trap is the pressing of a button. However, almost every type of control you can place on a form has the capacity to send out events. Events are messages from an object. For example, during a long car trip, the Sister object might signal the HesTouchingMe event.

This event is a message that some other object (the Dad object, perhaps) might respond to (maybe by invoking the StopTheCar() method). Events are signals to other objects that something has happened. Most of the controls are capable of sending dozens of different types of events. When you write a GUI program, you have to inform the system which events of which controls you want to trap for. You must also specify what should be done if an event occurs. As you might expect, the Designer does much of this for you, but you should still know what's going on.

Creating and Adding the Components

I designed this simple program by first imagining the form in my mind. (For more complex programs, I draw a sketch on paper or a chalkboard.) I then decided which tools from the Toolbox would best meet my needs. The label is an obvious choice for sending instructions and text (the Ouch!!! message) to the user. The button is also a good way to get input from the user.

In the Real World

Command buttons are part of graphical user interfaces purely for psychological reasons. They really are unnecessary from the programmer's point of view because their main purpose is to be clicked and nearly every control supplied with .NET has the capacity to be clicked. Although a user might think that buttons are very important because they do things, a programmer knows that it's actually the code that does the work, and the button's job is to be a familiar metaphor. When users see a button, they know that something should happen when they click it. Although something might also happen when users click something else, it isn't as obvious. Almost every form you make will have at least one command button on it. Never underestimate the power of psychology when it comes to designing your forms with components the user will understand intuitively.

Adding an Event to the Program

When you have a control to which you want to attach an event, you have two choices. You can double–click the component itself to get its default event, or you can choose from a list of other events. Most of the time, you will simply use the default event, because it is the most frequently called event of the component. For example, the most common event you trap for in a button is the click event, but in the form itself, you more often want to add code to the load event, which occurs as the form is loading into memory.

Most objects support several events, though, and double–clicking the control supplies only the most common event code. If you double–click a component in the Designer view, the editor automatically makes event–handling code for the default event of that component. If you want to use some other event of the control, change the Properties window so that it points to the events while your control is active in the Designer. Figure 6.15 shows the properties window displaying the events that are active. The Events tab on the Properties window looks like a lightening bolt. This tab is visible only when the Designer is visible on the screen.

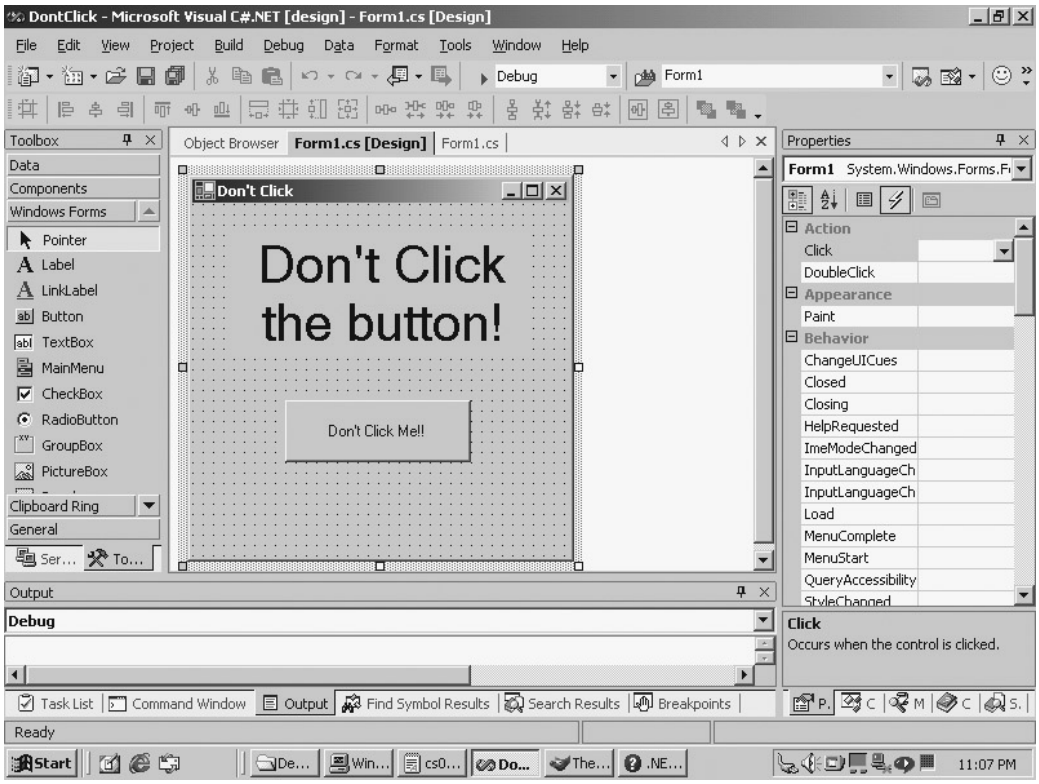


Figure 6.15: Double–clicking any event automatically creates an event handler for that event.

However you create an event, you are taken to a part of the code called an *event handler*. Event handlers are methods designed to occur when a specific event has been triggered. Here's the event handler I got when I double–clicked btnDont:

```
private void btnDont_Click(object sender,
System.EventArgs e) {

    }
```

Event handler names usually consist of the name of the object, an underscore (_) character, and the name of the event. All event handlers are automatically sent two parameters. The first is a sender object. This holds a reference to the object that sent the message. The second parameter is an EventArgs object. This special object is designed to hold other kinds of information about the event that just occurred. Generally, between the braces, you type the code you want to trigger. Here's how I caused the label to say "Ouch!!!" when the button is clicked:

```
private void btnDont_Click(object sender,
System.EventArgs e) {
      lblMessage.Text = "Ouch!!!";
    }
```

I copied the string value "Ouch!!!" to the Text property of the lblMessage object. Most of the input and output in a GUI program involves moving values to and from properties of controls.

Creating an Event Handler

The code for the Don't Click program was written almost entirely by the designer. Again, I want you to see this code in case you have to fix it.

The general code for creating this form is much like the Hello GUI program, so I won't show you the entire thing. Of course, the names of the specific controls were changed, but only a few other things

were added to make the program capable of responding to events. The following line was added to the InitializeComponent() method:

```
this.btnDont.Click += new
System.EventHandler(this.btnDont_Click);
```

The code tells the computer to call the btnDont_Click method as soon as the user clicks the button. This process is sometimes referred to as *registering* an event handler. C# requires the programmer to register all events that might occur in your program. Although this seems like a lot of extra work (Visual Basic 6 automatically recognizes every event of every component in your program), registering event handlers is a sound practice. First, it is usually done for you in C#. Second, only those events that contain code are registered. When the computer is running your program, it does not have to waste any time looking for events that have no code in them.

1In the Real World

The += operator might seem strange in this context because you are accustomed to seeing it used to add to some numeric value. C# has a feature called *operator overloading*, which allows an object creator to assign meaning to the ordinary mathematical operators when applied to this class. Programmers have had passionate debates about the wisdom and utility of operator overloading. I don't use it often because the meaning of mathematical operators can be vague if the class doesn't represent a number. In this particular case, it is not very intuitive to note that the += operator is adding an event handler to the click member of the button. However, this is a useful trick, and it is how event handlers are assigned in C#. Fortunately, you don't usually have to come up with this code from memory because it is created for you in the editor.

Allowing for Multiple Selections

In many situations, you want to have the user choose a value from a series of possible choices. The .NET framework supplies several controls that make it quite easy to add such features to your programs.

Choosing a Font with Selection Controls

The Font Chooser program, featured in Figures 6.16 and 6.17, demonstrates potentially useful selection tools. As a user, you've encountered radio buttons, check boxes, and list boxes. They are frequently used to allow the user to choose from a limited set of options. The Font Chooser program uses all these types of components to set the characteristics of a Font.

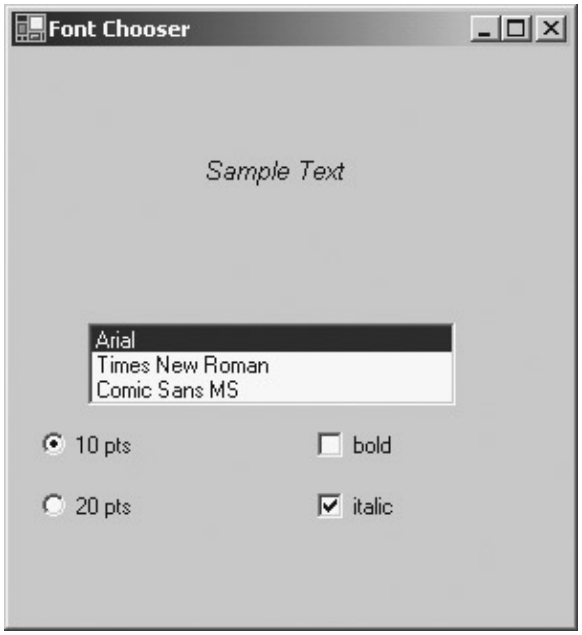


Figure 6.16: The user has used familiar interface tools to specify a 10–point Arial font with italic style.

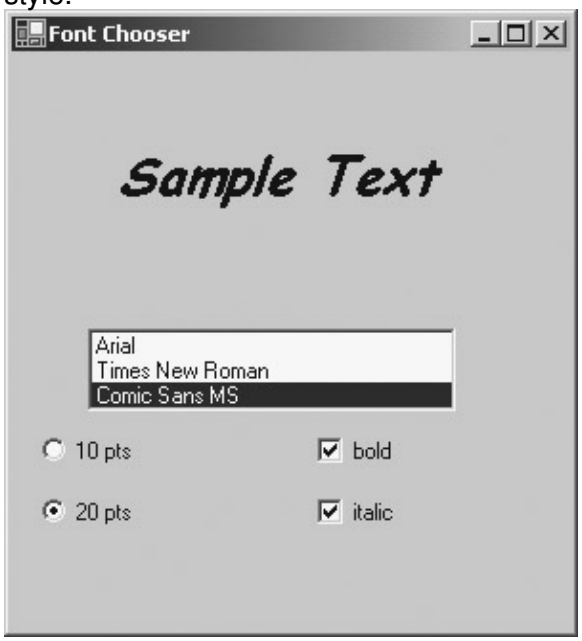


Figure 6.17: The user has selected a different font. Notice that the font can be both bold and italic but can have only one size.

Although more selection components are available to you, you will have no problems creating your own after you learn how the ones in this program work.

Creating the User Interface

Creating the visual interface is a logical starting point for any visual program, so that's where I began. Figure 6.18 illustrates the controls I used in this program. The form contains the following components:

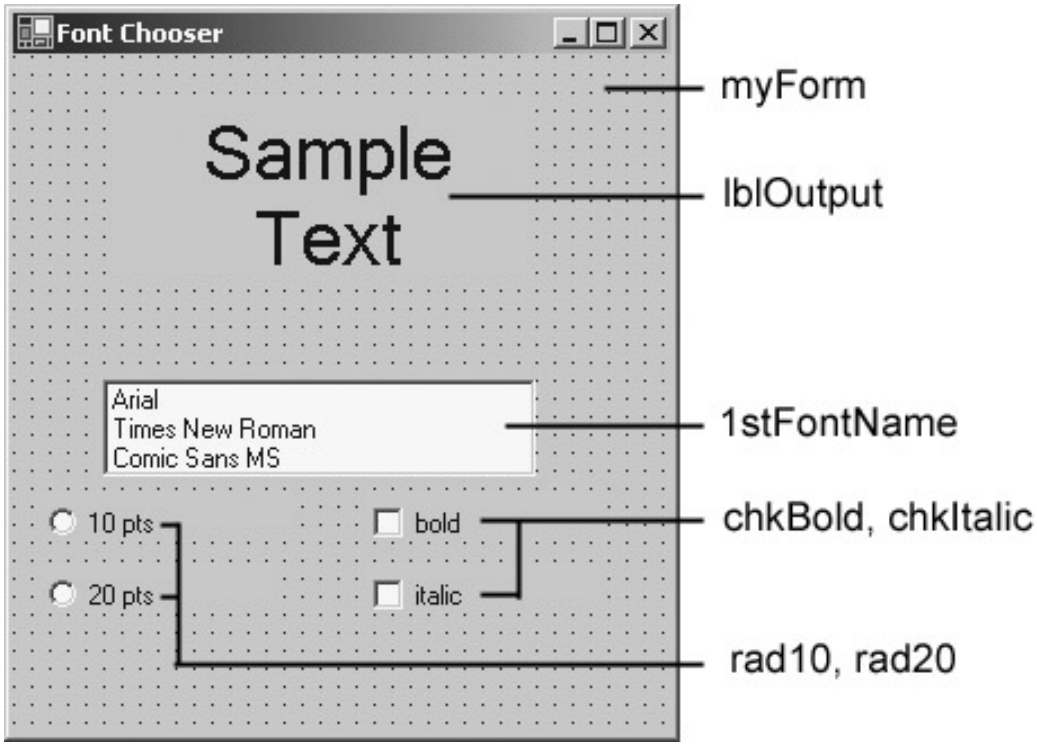


Figure 6.18: Sketch out the types and names of objects in your form.

- **myForm.** A form object
- **lblOutput.** A label that will show the output font
- **lstFontName.** A list box holding a series of font names
- **rad10, rad20.** Radio buttons to set the font size
- **chkBold, chkItalic.** Check boxes to set the font style

Start your programs by drawing up the forms and naming all the components. This step is often done best on paper when you aren't near a computer; then you can concentrate on the design itself. Give names to each element on the form, including the form itself. Also, the object names begin with a three−character abbreviation indicating the kind of control the object is. This is *Hungarian notation* and is widely used in GUI programming. Hungarian notation helps you remember what kind of control a variable name refers to. This eliminates the possibility of mistakes.

Hint Try to create your entire form before you begin adding code. Most importantly, change the names of all your components before you begin to write event handlers. If you change an object's name after you have written the event handler for that object, the event logic will no longer be associated with that control.

Trap The Visual Designer writes sloppy code for you if you let it. Be sure to name all objects before you write event handlers. Also, the Visual Designer sometimes crashes if you change the main form's name. It fails to change the Application.Run() line to the new form name, so you must make this change manually. Also, the file name of the form retains the old name, so you'll probably want to use the Save Form1.cs As option from the File menu to rename the actual file on the disk. I hope that these issues are simply problems with the beta version of C# that I had as I wrote this book and that Microsoft has corrected these errors by the time you write the programs. Still, this situation illustrates why you shouldn't trust the Visual Designer to build correct code for you. You need to know how the underlying code works because the Designer will make mistakes.

Examining Selection Tools

The .NET environment provides great interface tools for allowing multiple selections. I used three distinct tools—the list box, radio buttons, and checkboxes—that allow the user to select from a list of choices. Each has slightly different characteristics.

The List Box

A *list box* is a box that shows a list of strings. The list box is terrific when you have a large number of strings that you want the user to choose from. The list box is closely related to the combo box, which allows you to make a drop–down menu, instead of a list, where all elements are visible. Both the list box and the combo box feature the items property, which holds all the strings in the list box. If you highlight the items property in the Properties window when a list box or combo box is visible, a small button with an ellipsis (...) appears inside the Properties window. You can select this button to bring up an editor that allows you to enter the elements you want stored in the list box. Figure 6.18 illustrates editing the contents of a list box.

Radio Buttons

Radio buttons get their name from the old car radios that had pushbuttons. When you pushed one button to select a radio station, no other button could be pressed at the same time. (A radio can be tuned to only one station at a time.) In GUI designs, you use a set of radio buttons when you want to allow only one element from a set of elements to be active at a time. In the Font Chooser program, it makes sense for a font to have only one size at a time, so the font size is a perfect choice for a radio button group. All the radio buttons that live in the same container are considered part of a group. If you want to have more than one group of radio buttons on a form, you create a panel for each set of radio buttons and place the radio buttons on the panel.

In the Real World

Entire books have been written about effective user interface design. The design of interfaces is a complex field that touches on the disciplines of computer science, human factors engineering, cognitive psychology, design, and art. There is some debate about what goes into a good user interface design, but a few questions are clearly important in any design:

- Does the user interface help the user?
- Does the interface prevent illegal input?
- How well does the interface utilize the available space?
- Does the interface add to the theme and spirit of the software?

If you keep these questions in mind as you choose the types of components you put in your forms, you will build an appropriate interface for your program. If you are working on professional software, you can consult an expert on user interface design and usability testing. This ensures that the user interface of your software will help your program communicate clearly with your users.

Check Boxes

The font style is controlled by check boxes. You can set as many check boxes to true as you like. Check boxes are a good choice for the font style because a font can have any combination of bold and italic (or neither).

Creating Instance Variables in the Font Chooser

To write the Font Chooser program, I created the form design illustrated in Figure 6.18 and modified the program. Realizing that the key part of the program would be the creation of a font, I looked up the font object in the .NET documentation. Figure 6.19 shows the initial help screen for the font object.

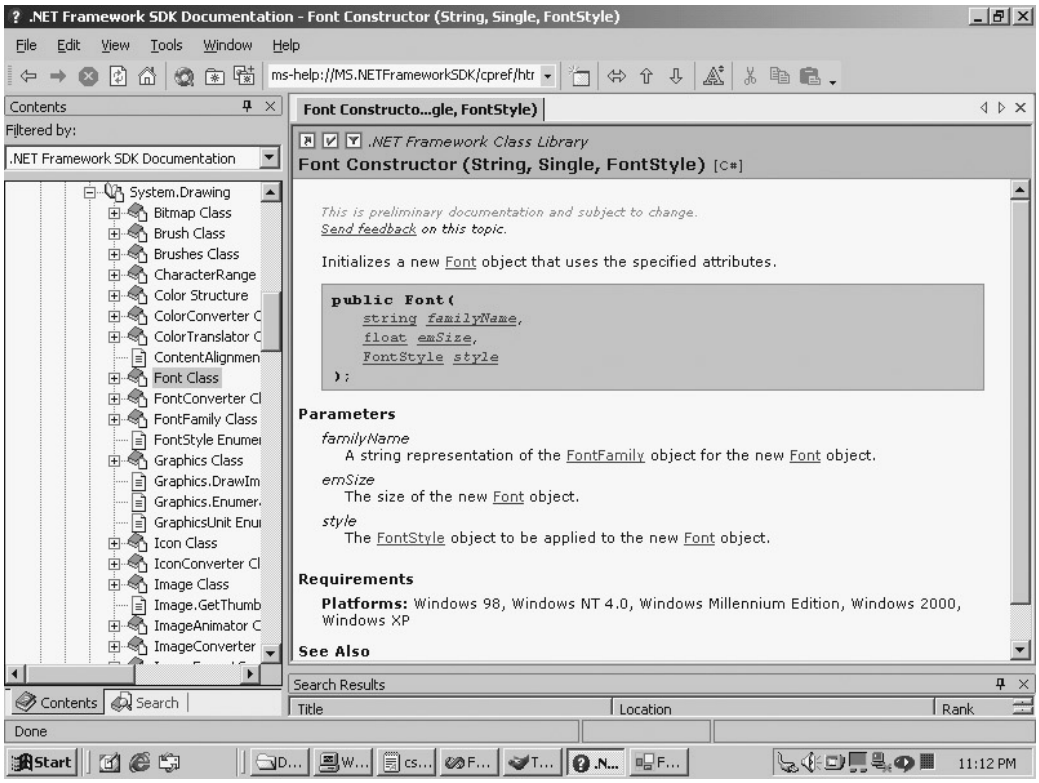


Figure 6.19: The Font class has a three–parameter constructor that builds a font based on exactly the information the form generates.

I found the class in the System.Drawing package. The font object has several constructors, but I was interested in creating a font based on the font's name, size, and style. Fortunately, I found a constructor that creates a font using exactly those parameters. I let the Designer build the initial code. Then I added some local instance variables for each part of the font. The following code generates my private instance variables for the font:

```
public class myForm : System.Windows.Forms.Form
{
  //Here's my own variables
  private System.Drawing.Font myFont;
  private string fontName = "Arial";
  private int fontSize = 20;
  private FontStyle myStyle = FontStyle.Regular;

  //these variables were created by the designer
  private System.Windows.Forms.ListBox lstFontName;
  private System.Windows.Forms.Label lblOutput;
  private System.Windows.Forms.CheckBox chkBold;
  private System.Windows.Forms.CheckBox chkItalic;
  private System.Windows.Forms.RadioButton rad20;
  private System.Windows.Forms.RadioButton rad10;
```

Note that I also included the instance variable declarations inserted by the Designer. Putting your own variables at the top of the class definition is better because the Designer always adds new

variables at the end of the instance variables list. You'll probably want to separate the variables you have created from those made by the Designer software. Notice, also, how I commented the variables to track which ones I created.

I knew that I would eventually create a style based on a string for the font's name, a number for its size, and an instance of the FontStyle object (whatever that is) for the style. Of course, I then looked up FontStyle and found out that it is a simple code listing all the legal styles of fonts.

In the Real World

Technically, FontStyle is an *enumeration,* which is a fixed list of values with names. However, you don't have to know this to use it. When you need it, you can do as I did and look it up to learn the special characteristics that make it useful. In the section " Getting the Style from the Check Boxes," I'll describe more fully how I used FontStyle.

Writing the AssignFont() Method

Whatever actions the user takes in this form will result in creating a new font object and assigning it to the Font property of lblOutput. I wrote a method named AssignFont() to handle this duty:

```
private void AssignFont(){
  // uses the variables to assign a font

  //check the list box for a font name
  fontName = lstFontName.Text;

  //look at check boxes for styles
  myStyle = FontStyle.Regular;
  if (chkBold.Checked){
    myStyle = myStyle | FontStyle.Bold;
  } // end if
  if (chkItalic.Checked){
    myStyle = myStyle | FontStyle.Italic;
  } // end if

  //create the new font and attach to the label
  myFont = new Font(fontName, fontSize, myStyle);
  lblOutput.Font = myFont;
} // end AssignFont
```

Trick To write a method that isn't associated with an event, write the method just as you do in ordinary classes. However, you might want to put your custom methods right after the instance variable declarations. All event methods created by the editor are placed at the end of the class automatically. Many programmers prefer to keep custom methods and automatically generated methods (the event handlers) in distinct segments of the file.

The key to the AssignFont() method is in the last two lines. These code lines create a new font based on the fontName, fontSize, and myStyle variables and copy that font to the Font property of lblOutput. The beginning part of the method gets the values from the various selection elements on the screen.

Getting the Font Name from the List Box

By the time the user sees the list box, it has a set of legal font names in place. When the user clicks a font name, the Text property of the list box reflects the currently selected value. A list box is perfect for a situation like this because font names are very specific. If you were to allow the user to type in a font name, you would have to do all kinds of error trapping in case the user either asks for a font that does not exist or makes a typing error. If you have more items in the list than can fit on the area allocated on the screen, the list box automatically adds a scroll bar. To retrieve the font name from the list box is a very straightforward affair:

```
fontName = lstFontName.Text;
```

This copies the currently selected text in the list box to the fontName variable.

Getting the Style from the Check Boxes

The font style is more complicated because there are several possible combinations of styles. Here's the code that generates a font style:

```
//look at check boxes for styles
myStyle = FontStyle.Regular;
if (chkBold.Checked){
  myStyle = myStyle | FontStyle.Bold;
} // end if
if (chkItalic.Checked){
  myStyle = myStyle | FontStyle.Italic;
} // end if
```

The segment begins by assigning the value Regular to myStyle. Check boxes have a Boolean property named Checked that returns true if the box has been checked by the user and false, otherwise.

If the chkBold check box's Checked property is true, the Bold style is incorporated into the style object. If the chkItalic is checked, the Italic style is incorporated into the style object. I incorporated the styles by using a little binary magic. The pipe symbol (|) indicates a *bitwise or* operation. The values of the various styles are set up so that you can use this operator to combine any number of values. You can check with your friendly neighborhood computer scientist to discover what a bitwise or operator does and why it is used here, or you can simply trust the documentation that comes with the FontStyle entry in .NET (as I did) and move on. (Okay, I *am* a friendly neighborhood computer scientist, but you really can live without all those details right now.)

The end result of this code fragment is that myStyle will be bold if chkBold is checked, italic if chkItalic is checked, bold and italic if both check boxes are selected, and neither bold nor italic if neither box is checked.

Finding the Font Size Code

You might be surprised that there is no code in the AssignFont() method for dealing with setting the font size. It turned out to be more convenient to assign the size when the option button is pressed. You'll see the code when I show you the event handlers (next). I'll also explain why I didn't put that code in AssignFont().

Writing the Event Handlers

The event handlers for the Font Chooser are very simple. Essentially, all of them defer control to the AssignFont() method:

```
    private void lstFontName_SelectedIndexChanged(object sender,
System.EventArgs e) {
      AssignFont();
    }

    private void chkBold_CheckedChanged(object sender,
System.EventArgs e) {
      AssignFont();
    }

    private void chkItalic_CheckedChanged(object sender,
System.EventArgs e) {
      AssignFont();
    }

    private void rad10_CheckedChanged(object sender,
System.EventArgs e) {
        fontSize = 10;
        AssignFont();
    }

    private void rad20_CheckedChanged(object sender,
System.EventArgs e) {
      fontSize = 20;
      AssignFont();
    } // end AssignFont
```

I started to write complete code for each event, until it became clear that the event methods would be almost identical. Any time you find yourself writing repetitive code is a good time to consider encapsulation, so I moved all the code that is common to all the methods to the AssignFont() method and had each event handler call that method. If I decide later to change my logic, I'll have to change it in only one place. The radio buttons are the only event handlers that have additional code. If a radio button is pressed (which is what causes the CheckChanged event to trigger), that button's checked property is true, and all the others are false. I set the fontSize variable in the event handler because in the event handler you know what the correct size will be. If you wait for the AssignFont() method to check for the font size, you have to check each radio button to see which was pressed.

I could not use this technique in the check boxes because the state of the font style depends on *all* the checkboxes, not just the currently selected one.

Working with Images and Scroll Bars

The selection elements you learned in the preceding section are great when you want some kind of text input from the user. However, often you are interested in numeric or spatial information from the user. Also, if the interface is graphical, you should be able to incorporate custom graphics into your programs. In this section you will learn how to do both these things.

Changing an Image's Size

The Sizer program demonstrates how to incorporate custom graphics into your C# programs using the picture box control. It also illustrates the scroll bar control, which is handy when you want the user to choose an integer value from a specified range (even if the user doesn't know what an integer is). Figures 6.20 and 6.21 demonstrate the Sizer in all its impressive glory.

Figure 6.20: In its default state, the picture is small, and the scroll bar is all the way to the left.

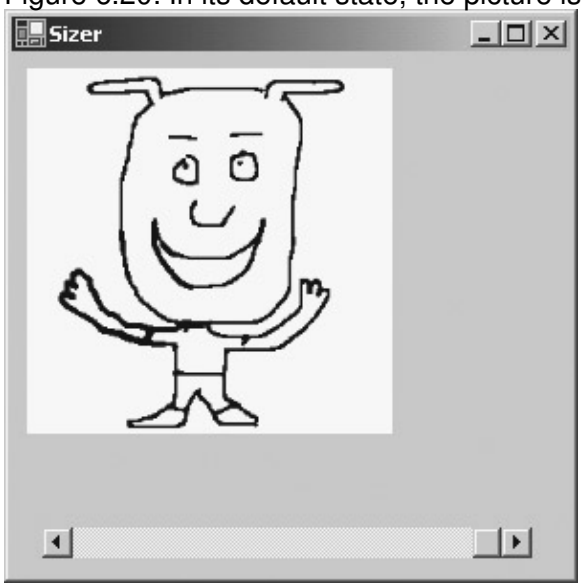


Figure 6.21: When the user drags the scroll bar, the image becomes larger.

This program has only one line of custom code. Most of the work went into setting up the form. Figure 6.22 shows my diagram of the form and its objects.

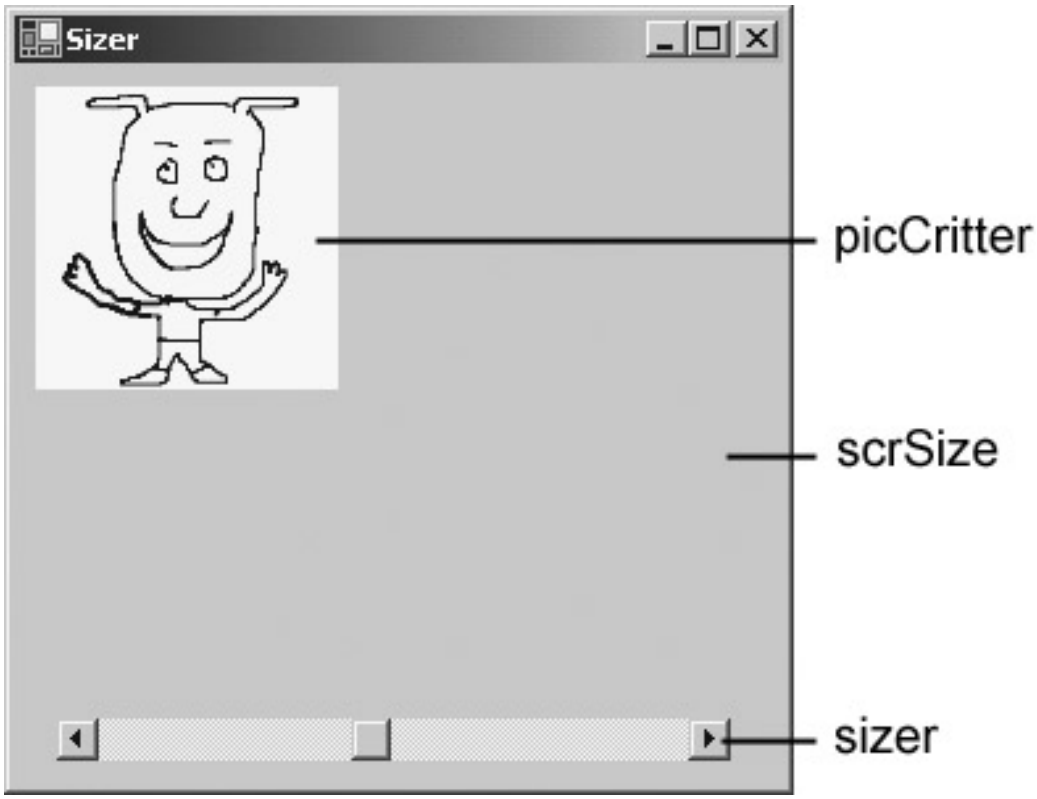


Figure 6.22: My initial sketch of the Sizer program.
When I had some idea of what I wanted to do, I could find the right objects to build the form and the right properties to manipulate the objects.

Setting Up the Picture Box

The picture box control is cleverly named because it is a box that holds a picture. It is a very convenient way to add graphics to your programs. You can create an image in any image–editing software and attach that image to the picture box control. You can then manipulate the image by adjusting the control's properties. The image property is used to assign an image to the picture box. Picture boxes can take many standard image types, including Windows bitmaps (.bmp files), Graphical Interchange Format (.gif) files, and Joint Picture Experts Group (.jpg) format files. Generally, I like to use a compressed format such as .gif or .jpg, but the choice is up to you.

Trick If you are going to be doing much graphics work, you need a powerful image–editing tool. Many quality commercial packages are available, but in case you're on a budget, an excellent freeware package called the *Gimp* is included on the CD–ROM accompanying this book. The Gimp is an open–source program originally written for the Linux family of operating systems, but the version included on the CD works in Windows. This program features most of the high–end tools you need in an image editor, including layers, transparency, channels, masking, paths, Bèzier selection, and so on. I used it to prepare all the screen shots for this book.

Adding an Image to the Picture Box

You can add an image to the picture box by clicking the image property in the Properties window while a picture box control is selected. You will see the ellipsis button indicating that a special editor is available for initializing the property. When you click the ellipsis button, you see a dialog box that lets you pick an image file from your hard drive. You can use an image stored in the most common image formats used in Windows and on the Web, including raster formats such as .bmp, .gif, .jpg,

and .png and the vector formats .wmf and .emf. You can use an image you already have access to or create your own.

In the Real World

Many images are copyrighted, so you must get permission before using an image somebody else has created. If you plan to release your software, get permission to use art work, create it yourself, or purchase it (select a finished product or hire an artist). Most of the time, I create crude graphics, and if they need to be improved (they usually do), I bring in a professional artist. Still, it's good to have sketches so that you can indicate what you're trying to accomplish.

Manipulating the Image's SizeMode Property

Images almost never come in the size you need for your programs. If your picture box is exactly the same size as the picture it is showing, everything is fine. This rarely happens, so the SizeMode property of the Image object gives you several ways to resolve this conflict.

Figure 6.23 demonstrates the various settings of the SizeMode property.

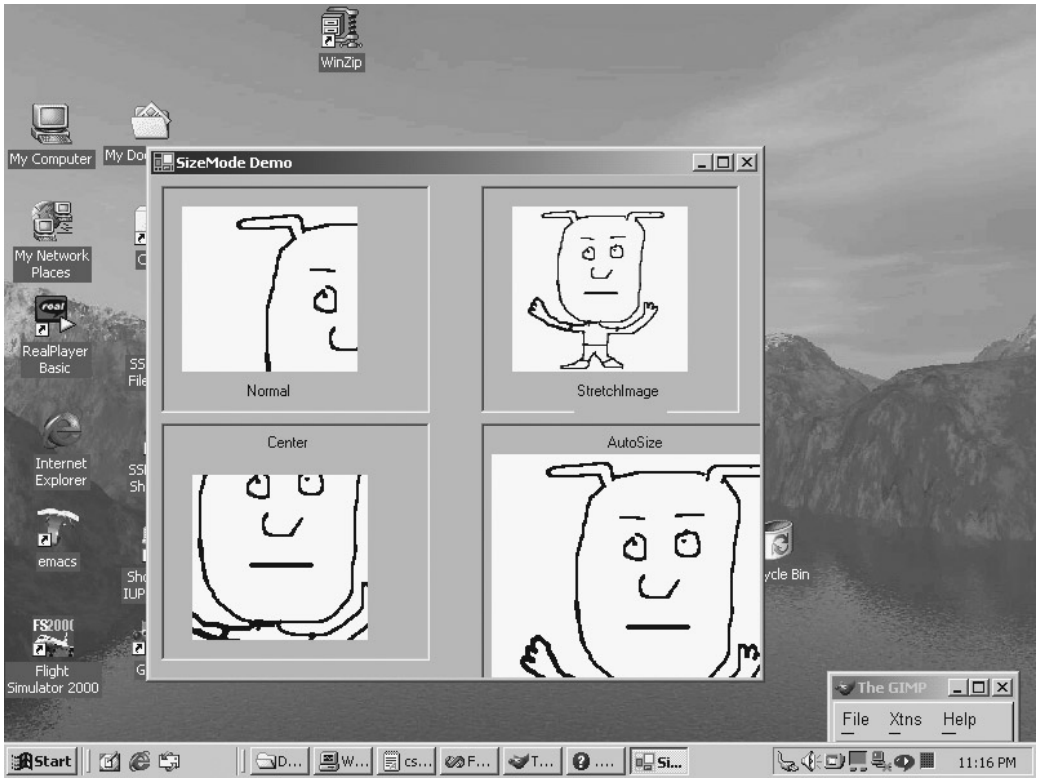


Figure 6.23: All the picture boxes are the same, except for the setting of SizeMode.

Trick I did not include a source code listing for this program because I added no event logic at all. The visual layout of the program illustrates what I'm describing in this section. The actual program with source code is available on the CD–ROM, so you can look at it there.

The default setting for the SizeMode property is Normal. The image will be loaded into the picture box at its normal scale, but if the picture box is too small, the bottom and right segments of the image will be trimmed. Setting SizeMode to StretchImage causes the image to be stretched and squashed so that it fits the size and shape of the picture box. This can cause distortion if the image and the picture boxes are not the same general shape. For example, if the image is square and the

picture box is a tall skinny rectangle, the image will appear to be taller and skinnier than usual. The Center setting causes the image to be displayed at its default size but centered within the image box, with the edges cropped out. The AutoSize setting causes the picture box to shrink or grow so that it is exactly the right size for the image.

Adding a Scroll Bar

Scroll bars are a wonderful innovation in interface design. Many times, you need some sort of integer input. If you ask a user to type in a number, you never know exactly what you will get. You usually have to do all sorts of error–detection gymnastics to ensure that the number is properly formatted and is not written out. Also, users generally prefer to use some sort of visual interface if they are in a GUI. The scroll bar control is designed to let the user visually enter a number. Much of the time, the user isn't even aware that this is what he or she is doing, but that's what scroll bars are for. The .NET framework supplies two scroll bar controls, but they are almost identical. The HScrollbar is aligned horizontally, and the VScrollbar is vertical. (Not surprisingly, both are inherited from a generic Scrollbar class.) The little box inside the scroll bar is sometimes called the *elevator.* The Value property of a scroll bar is an integer related to the position of the elevator. For horizontal scroll bars, smaller values are on the left. For vertical scroll bars, smaller values are at the top. You can set a range of values your scroll bar will return, with the Maximum and Minimum properties. Also, you can change how much the elevator moves when the user clicks the arrowheads, by setting the SmallChange property. You can indicate how far the elevator moves when the user clicks the shaft, by adjusting the LargeChange property. For the Sizer program, I chose a horizontal scroll bar with values between 50 and 200 because I want the image to vary between 50x50 and 200x200. I set the Minimum property to 50 and the Maximum property to 200. I left everything else alone.

Writing the Event–Handling Code

This program requires only one line of custom code, and that goes in the default event handler of the scroll bar. When the scroll bar is changed, the picture box's size should also change to have the same height and width as the scroll bar's value. Here's the event code:

```
private void scrSize_Scroll(object sender,
System.Windows.Forms.ScrollEventArgs e) {
      picCritter.Size = new Size(scrSize.Value,
scrSize.Value);
    }
```

The Scroll event occurs whenever the user moves the scroll bar. When this happens, the program changes the Size of picCritter. The Size property requires a size object (I learned that by looking at the object browser for the picture box). I looked up the size object and found that I could create it with two integers. Because I wanted the picture box to remain square, I just set the value of the scroll bar as both the X and Y values for the new size object.

Revisiting the Visual Critter

The Visual Critter program mentioned at the beginning of this chapter combines all the techniques you've learned throughout the chapter—and adds a few minor twists. The visual Critter is a picture box. The user changes the visual critter's characteristics by manipulating the various controls on the screen.

Designing the Program

As a starting point, look at the sketch I used to design the program in Figure 6.24.

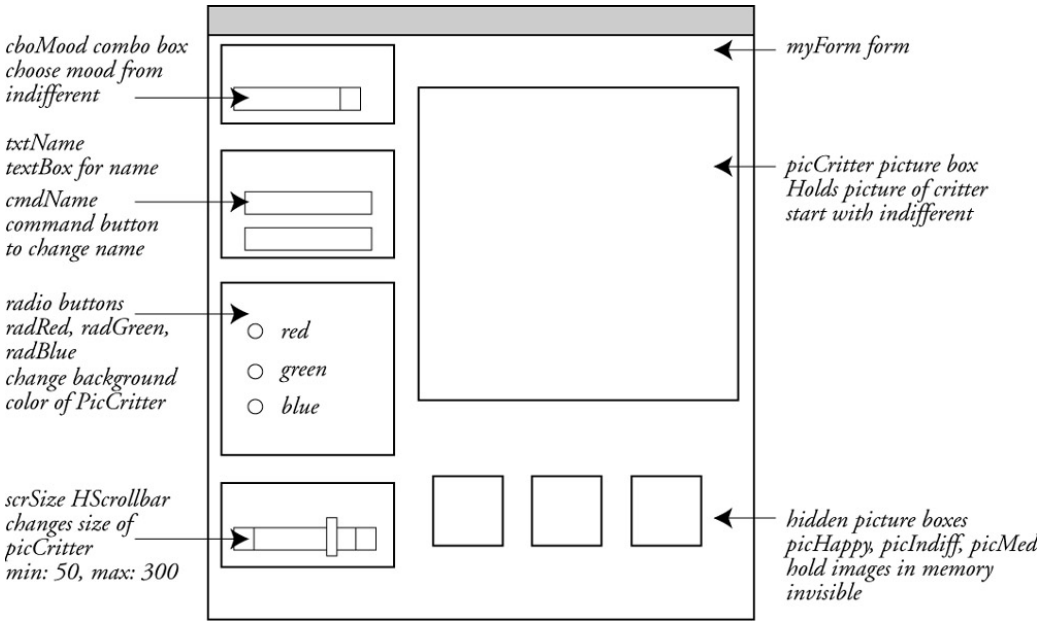


Figure 6.24: This sketch helped me to define how my program would be built.

Because the visual interface is so critical to GUI programs, the statement of the problem almost always involves at least one sketch of the user interface. (If you prefer not to sketch, your notes can be in English.) In visual programs, it's a very good idea to figure out how the screen layout will be designed and determine the main types of objects you expect on the screen. In this type of sketch, I also like to name all my objects and write down any special characteristics of the objects. For example, you can see that I have predicted the range of the scroll bar and have indicated that several of the picture boxes will be invisible. The point of the sketch is not to solve all the programming problems. The goal is to make sure that you know what you're trying to accomplish before you start writing code. Changing your ideas on paper is much easier than making changes after you start writing the program.

From the sketch, you can see that the program is centered around a picture box named picCritter. All the other elements on the form modify picCritter. I have decided to let the user change picCritter in four ways: The user can change the critter's mood (which will change the image shown in picCritter) with a special form of the list box called the *drop–down list box*. The user can change the critter's name by typing in a text box and clicking a button. The critter's color can be modified by selecting a new color from a set of radio buttons, and the size is changed through a scroll bar. While I was sketching this out, it became clear that the number of controls on the form would be overwhelming, so I drew boxes around various elements to group the similar controls. For example, all the radio buttons are in a box, as are the text box and command button that deal with the critter's name. I also added three special picture boxes that will be invisible to the user. I'll explain soon what those are used for.

In the Real World

Many programmers design their programs first on chalkboards or whiteboards because these surfaces allow for even more flexibility than paper. A surprising number of my programs have been designed on napkins because a good idea came to me while I was at a restaurant. It doesn't matter how you write your sketch (until you become a professional, when there will undoubtedly be standards), but you should do your initial work away from the programming environment. Resist the

temptation to design your form on–the–fly in the Designer. The .NET Designer will become confused if you change the design and layout too many times. It's better to start the editor when you are clear about what you want your program to do.

Determining the Necessary Tools

Almost all the tools necessary for this project are described throughout this chapter, but my sketch pointed out the need for a couple variations.

A list box is a nice control but takes up a lot of room on the screen. What works better is the *combo box*, which is a form of the list box that expands when needed but takes up less room on the screen when the user is not using it. The combo box is a souped–up list box. It has all the same features as the traditional list box. Also, it can be set up so that the user can type in a value or set up as a list box with drop–down behavior. You set this behavior by modifying the comboBoxStyle property. A drop–down list box will be a nice addition to the program because it allows the user to pick from several options but doesn't take up too much room on the screen.

In the Real World

Drop–down list boxes are popular with programmers because they allow many choices without taking as much space on the screen as check boxes, radio buttons, or traditional list boxes. Space is crucial when more items might be added to the list of choices. The other types of controls require space for new options. With a drop–down list, you can always add more elements to the list without requiring any more screen real estate. The other style of combo box is handy when you want to let the user type in a value or choose from a list.

The form has many controls on it, so breaking up the form into smaller segments is necessary. By digging around in the Toolbox, I found the panel control, which is perfect for this kind of thing.

I also needed to store the various images for the critter temporarily. I simply created extra picture boxes and set the visible property to false. The picture boxes are then invisible, but you can still copy the image property over to change the image of the picCritter picture box.

Designing the Form

After I figured out how the project should be laid out, I built the form. Figure 6.25 shows the form as it is being built in the Designer.

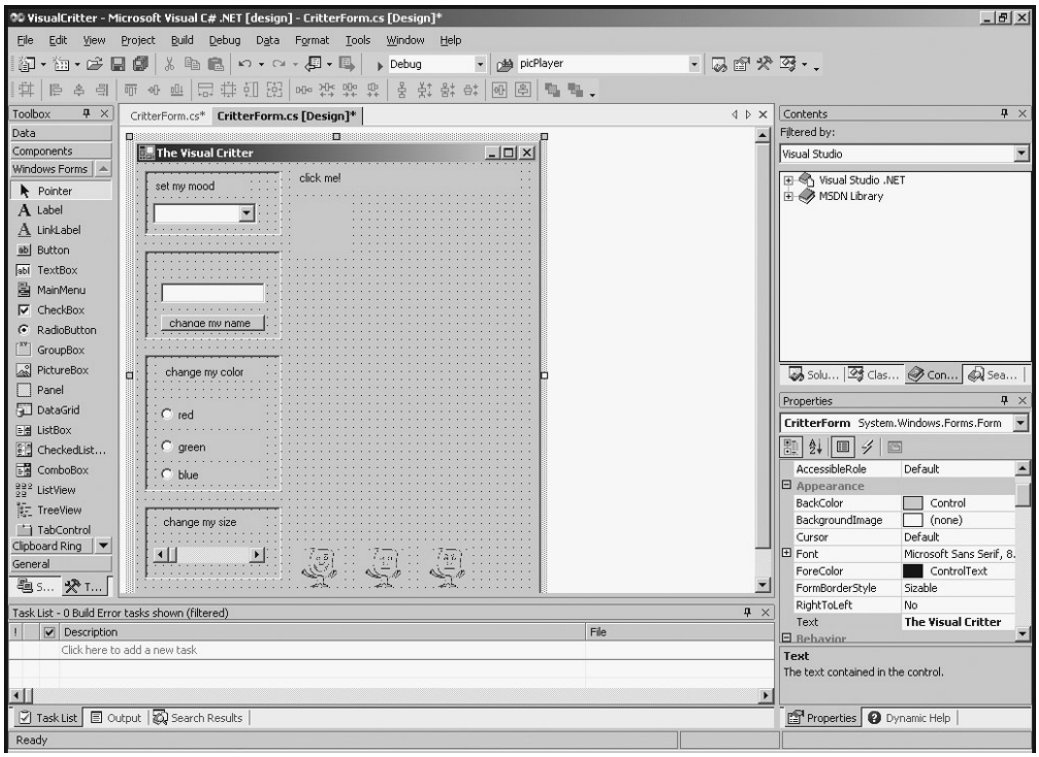


Figure 6.25: The form is easy to build if you have a sketch to follow.
I built the form by following the sketch. Notice that I added some panels to group the various types of controls and added a few labels to help the user understand what's going on. I changed the panel's border property to get the 3–D effect.

Trick When you use panels to separate controls, be sure to draw the panel first and then draw the control inside the panel. If you draw the control first, it will be attached to the form itself, rather than to the panel. If you draw the control inside the panel, you can move the panel, and all the controls will move with it. Also, you can use panels to isolate groups of radio buttons if you have more than one group of such controls on your form.

I also added one label above the critter image to handle displaying the critter's name. In the original plan, I didn't have a way for the critter to tell its name.

Writing the Code

In this type of program, most of your code consists of anticipating user actions and writing methods to handle events. All the events are the default events of the objects.

Adding Instance Variables

The Visual Critter program requires only one additional instance variable. I added myName as a string variable to hold the critter's name. As usual, the instance variable goes inside the class definition line and before any method definitions.

Responding to a Change in the Combo Box

The combo box cboMood will fire off a SelectedIndexChanged event whenever the user chooses a new element from the combo box. If that happens, the program copies an image from one of the hidden image boxes.

```
private void cboMood_SelectedIndexChanged(object sender,
System.EventArgs e) {
      switch (cboMood.Text){
        case "happy":
          picCritter.Image = picHappy.Image;
          break;
        case "indifferent":
          picCritter.Image = picIndiff.Image;
          break;
        case "mad":
          picCritter.Image = picMad.Image;
          break;
        default:
          picCritter.Image = picIndiff.Image;
          break;
      } // end switch

    }// end cboMood event
```

The Text property of cboMood will contain whatever value is selected in the combo box. A switch statement examines all the possible values and copies the Image property of the appropriate invisible picture box to the image.

In the Real World

You can select and store images in several ways, but this is one of the simplest. Having the images already loaded in memory in hidden controls is preferable because loading an image from the disk drive can be a slow process. If the image is already in the computer's memory (as it is when assigned to an invisible picture box), the image can be copied very quickly to the visible picture box.

Notice that I added a default clause, even though it should never happen. You should add a default clause to every switch statement because strange things happen and it's better to be safe than sorry.

Clicking the Critter

When the user clicks the critter, it should say something. Here's the code that occurs when the user clicks picCritter:

```
private void picCritter_Click(object sender,
System.EventArgs e) {
      if (myName == ""){
        lblName.Text = "Please give me a name!";
      } else {
        lblName.Text = "My name is " + myName + "!";
        MessageBox.Show("Hi, my name is " + myName);
      } // end if
    } // end picCritter click
```

If the myName variable hasn't been changed by typing a new name in the text box, the critter asks the user to supply a name. Otherwise, the program copies a friendly message to the lblName label. Notice the line that starts with MessageBox. The message box object is a convenient way to send a quick message to the user. Use the MessageBox.Show() method to say something in a little dialog box.

Trap The MessageBox.Show() method is very handy but interferes with the flow of the program. Also, if you use it too much, your users will be annoyed. Integrating all your output into the form (when you can) is much more in the spirit of GUI programming. Also, if you're familiar with programming in Visual Basic or JavaScript, you might wonder where the equivalent to inputbox or prompt is in C#. The .NET framework does not supply a quick dialog for input. You have to make your own if you want one. Generally, though, you can get all the input you need from your programs.

Changing the Critter's Name

The txtName text box and the btnName command button work together to let the user change the critter's name. You might be surprised that no code is attached to the text box. All the code for name changing happens in the command button. Here is the code:

```
private void btnName_Click(object sender,
System.EventArgs e) {
  if (txtName.Text == ""){
    MessageBox.Show
    ("Please enter a name in the text box
and click the button again");
  } else {
    myName = txtName.Text;
    MessageBox.Show("OK, You changed my name.");
  } // end if
} // end btnName click
```

This method ensures that there is a name in the text box. Then it copies that name to the myName variable and lets the user know that the name has been changed.

Changing the Critter's Color

The critter's color is changed through the radio buttons. The code is very straightforward:

```
private void radRed_CheckedChanged(object sender,
System.EventArgs e) {
      picCritter.BackColor = Color.Red;
    } // end radRed event

    private void radGreen_CheckedChanged(object sender,
System.EventArgs e) {
      picCritter.BackColor = Color.Green;
    } // end radGreen event

    private void radBlue_CheckedChanged(object sender,
System.EventArgs e) {
      picCritter.BackColor = Color.Blue;
    } // end radBlue event
```

Each of the radio buttons immediately changes the background color of picCritter to the appropriate color. Notice that all the legal color values are available in the System.Drawing namespace. If you type **Color.** while this namespace is available, you see a complete list of color names you can use.

Changing the Critter's Size

The Critter's size is changed by a scroll bar. In fact, I stole the code directly from the Sizer program featured earlier in this chapter.

```
private void scrSize_Scroll(object sender,
             System.Windows.Forms.ScrollEventArgs e) {
      picCritter.Size = new Size(scrSize.Value,
scrSize.Value);
    }
```

I changed the size property of picCritter to a new Size object with both a width and height equal to the current value of the scroll bar.

Summary

In this chapter you applied the OOP skills you have learned so far in this book. You have made the transition to Windows programming. You have learned how to build an interface using the IDE, but you have also learned what the Form Designer does as it creates code for you. You have been introduced to the very useful System.Windows.Forms and System.Drawing namespaces. You have learned how to set up the most critical types of controls on your forms, including text boxes, command buttons, scroll bars, and various kinds of selection devices. More importantly, you have learned how to explore the capabilities of any controls available in the Toolbox so that you can adapt to new controls when they are made available.

Challenges

- Add new capabilities to the Visual Critter. Perhaps have it say something random each time it is clicked, based on its mood.
- Integrate the visual interface with the critter object you wrote in Chapter 4, “Objects and Encapsulation: The Critter Program.” Use buttons to feed and play with the critter, and have the mood change (based on the hunger and happiness variables), with visual indicators of the critter’s satisfaction.
- Design a visual interface for the Snowball Fight program from Chapter 5, “Constructors, Inheritance, and Polymorphism: The Snowball Fight.” Use buttons to throw snowballs, make snowballs, and move.
- Design a graphical adventure story like the one in Chapter 2, “Branching and Operators: The Math Game.” Use text boxes, radio buttons, and other design elements to get user input.
- Make a version of the Rock, Paper, Scissors children’s game. The user and the computer each choose between rock, paper, and scissors, and the computer determines a winner according to a simple formula: Rock breaks scissors, scissors cut paper, and paper covers rock. Let the user click buttons or picture boxes to make his or her choice, and have the computer choice selected randomly.

Chapter 7: Timers and Animation: The Lunar Lander

When you move to a graphical interface, you can write many new kinds of programs. Games and animations are especially interesting to write. In this chapter, you will write a simple arcade game, with all the features you might expect from such a game. Along the way, you'll master techniques used for writing animations in C#. In this chapter you will learn how to

- Read individual key presses.
- Work with multiple images.
- Use the timer object to simplify animation.
- Move objects around on the screen.
- Detect collisions between objects.

Introducing the Lunar Lander

The sample game for this chapter re–creates one of the all–time classic arcade games, the Lunar Lander game. In this game, the user tries to pilot a spacecraft near the surface of the moon. The pilot must land his or her craft on a small platform with limited fuel. Figure 7.1 demonstrates the Lunar Lander game in action.

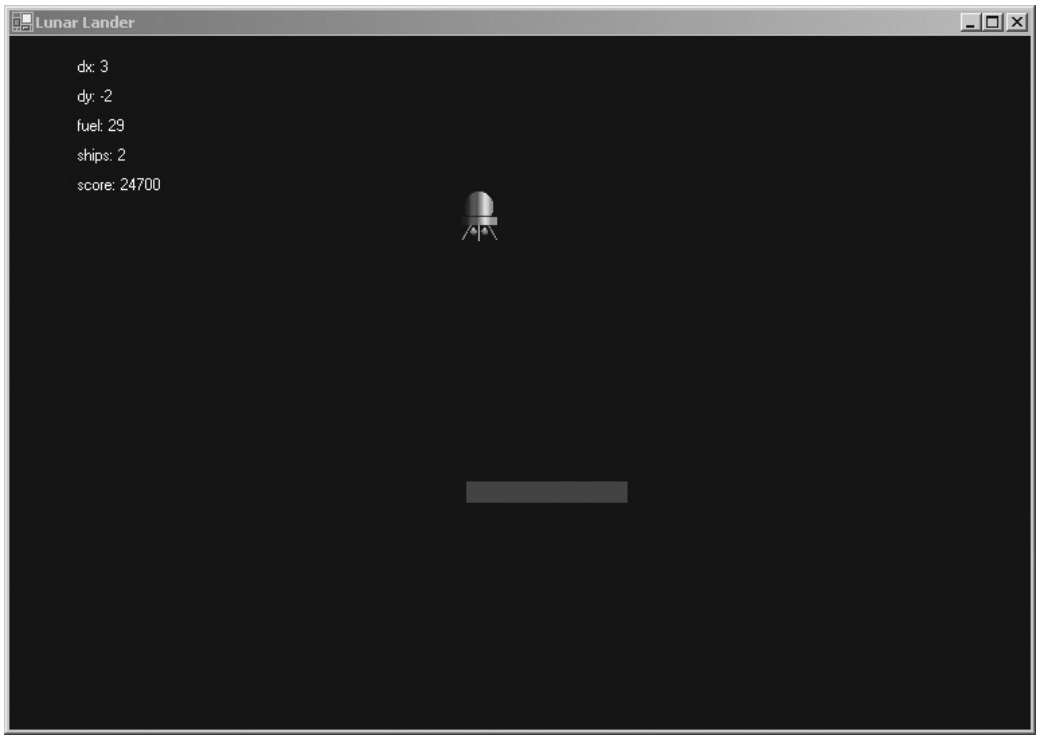


Figure 7.1: The landing craft starts with a random speed and direction—and limited fuel.

The player controls the ship with thrusters controlled by the arrow keys. However, each blast of a thruster burns precious fuel. If the player is going too fast when he or she touches the landing pad, it is a crash, and he or she loses a ship. A safe landing is rewarded with more fuel and a new, randomly chosen landing assignment. The game continues until the user loses three ships.

Reading Values from the Keyboard

To code the Lunar Lander game, you first need some way to capture keystrokes from the user. You can always use text boxes to read from the user, as you did in Chapter 6, “Creating a Windows Program: The Visual Critter.” However, in a game environment, a text box is distracting. Also, text boxes are generally concerned with entire phrases of text and do not handle special keys such as arrow keys and function keys well. Fortunately, C# provides features that make it easy for your programs to read input directly from the keyboard.

Introducing the Key Reader Program

The form object features three events that can be used to determine which key a user has pressed. The KeyDown event fires whenever a key is being held down. The KeyUp event is triggered when the user releases a key. The KeyPressed event occurs after the key is pressed and released. The keyboard events are used to look at the keyboard in different ways. The Key Reader program shown in Figures 7.2 through 7.4 demonstrates some of these differences between the KeyDown() and KeyPressed() methods. Look at the program first, and I’ll explain why the various responses seem to disagree.

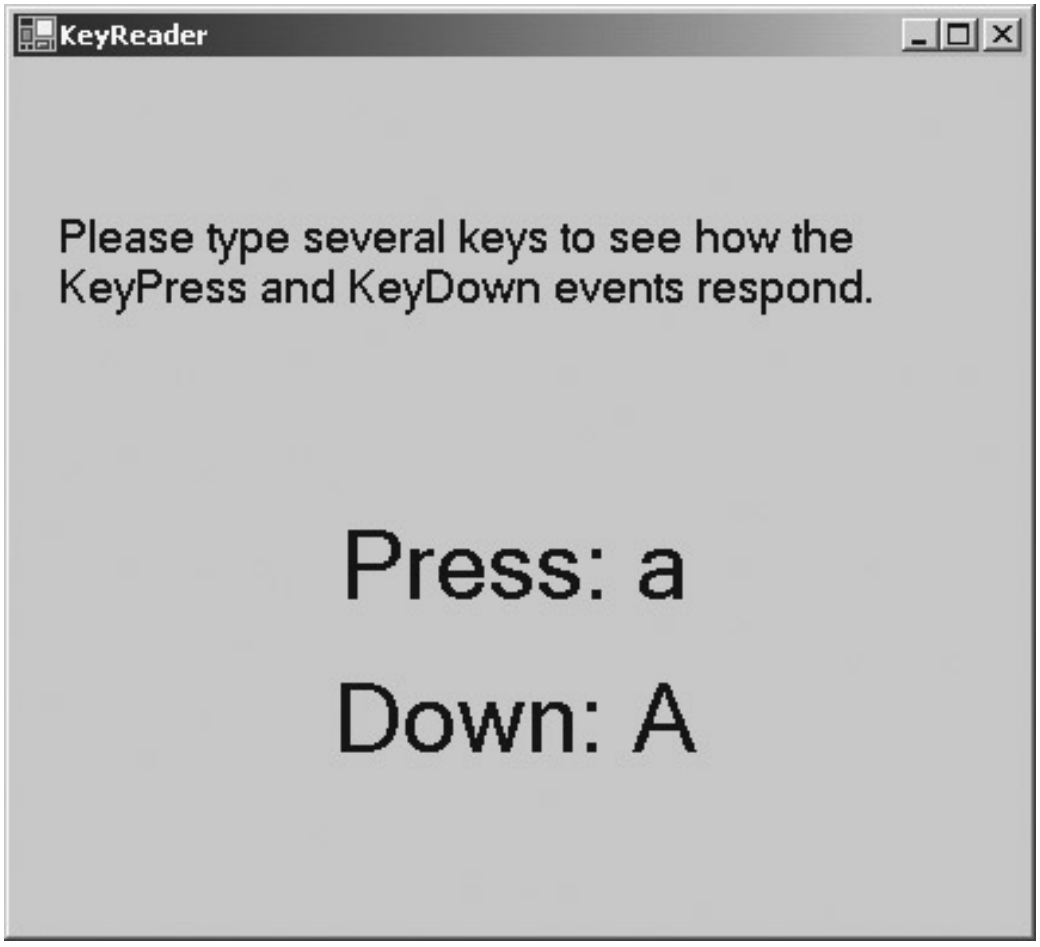


Figure 7.2: When the user presses lowercase a, both events display the value a, but they disagree on capitalization.

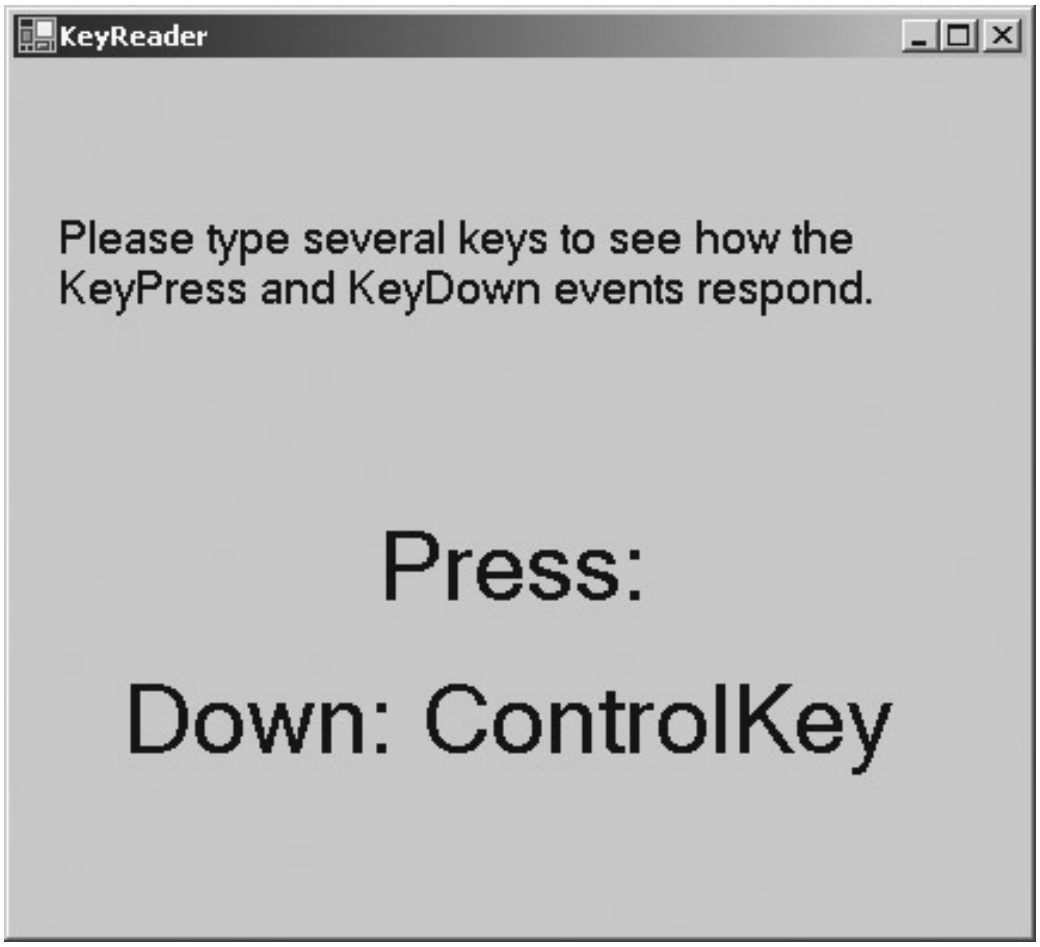


Figure 7.3: If the user presses a control key, only the KeyDown event can interpret the value.

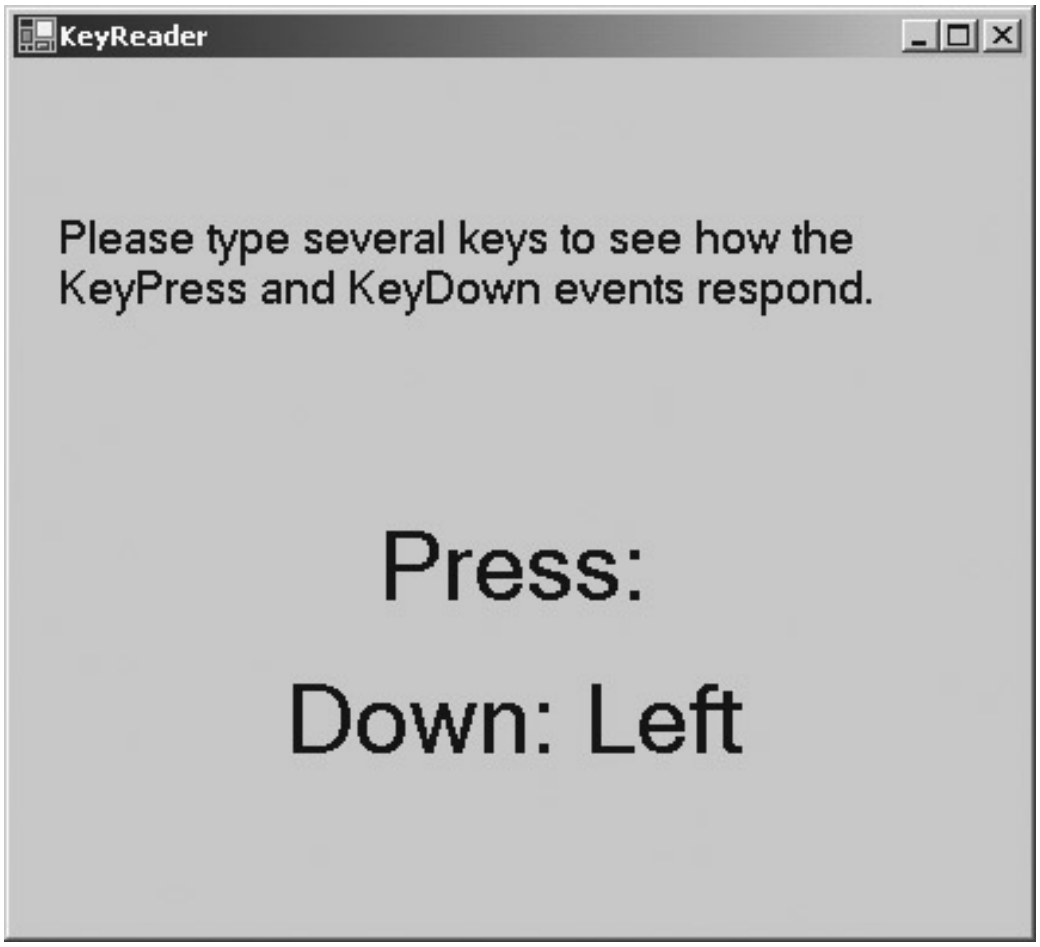


Figure 7.4: The KeyDown event can also respond to arrow keys, function keys, and other special keyboard inputs.

Setting Up the Key Reader Program

The setup of the Key Reader program is very straightforward. The visual layout of the program includes three labels. I named two of the labels lblDown and lblPressed. These two labels show information from the KeyDown and KeyPressed events, respectively.

Trick I didn't rename the third label; instead, I left it as the default name suggested by the Designer (label1). Although renaming is never a bad idea, there are a few objects you don't need to rename. If you have a label meant entirely for displaying text to the user, if that label will never have any events activated and you will never refer to the label, you don't need to worry about giving it a name. All these conditions are true for the label that holds the instructions to the user, so I didn't give it a more explicit name. If you don't want to worry about these rules, just rename every object in your programs, and you'll be safe.

There are no other components on the form. I did set one of the form's properties, however. The KeyPreview property determines whether a form will check keystrokes that are being sent to components on that form. For example, if you have a text box on the form, you might not want to trap for keyboard input when that text box is currently selected. Set KeyPreview to true to ensure that all keyboard presses are sent to the form handlers you will be writing shortly.

Coding the KeyPress Event

The KeyPress event occurs after the user has pressed and released a button on the keyboard. You

usually trap for the KeyPress event when you want to know which of the alphanumeric keys was pressed. The KeyPress event is not the default event for the form object, so to write code for it, you need to be sure that the form is visible in the Designer. Then choose the events list (with the lightning icon) in the Properties window. You see a list of properties the form recognizes. The KeyDown, KeyUp, and KeyPress events are listed on this screen. Choose the KeyPress event to code first, because it is the simplest. Here's the code produced by the Designer:

```
private void KeyReader_KeyPress(object sender,
         System.Windows.Forms.KeyPressEventArgs e) {
}  // end KeyPress
```

The KeyReader_KeyPress() method generated by the Designer provides two objects. The first is an object named sender, which refers to the object that triggered the event. Most event handlers have this parameter, although you will not need it the projects for this chapter.

Trick You can use sender if several buttons call the same event handler. Inside that event handler, you can do different things, based on which button called the event.

The other parameter is named e and is extremely useful. The e parameter is an instance of the KeyPressEventArgs class. It stores interesting information about what kind of keyboard event just occurred. Most event handlers pass an object as a parameter. Different kinds of event create different objects, but all of these event parameters are used to provide information about the event. The event objects are usually called e. Often, your event–handling code starts by looking at some parameters of e. Figure 7.5 shows the main help screen for KeyPressEventArgs so you can get a feel for its capability. Of course, you'll want to look it up in the documentation as you program with it.

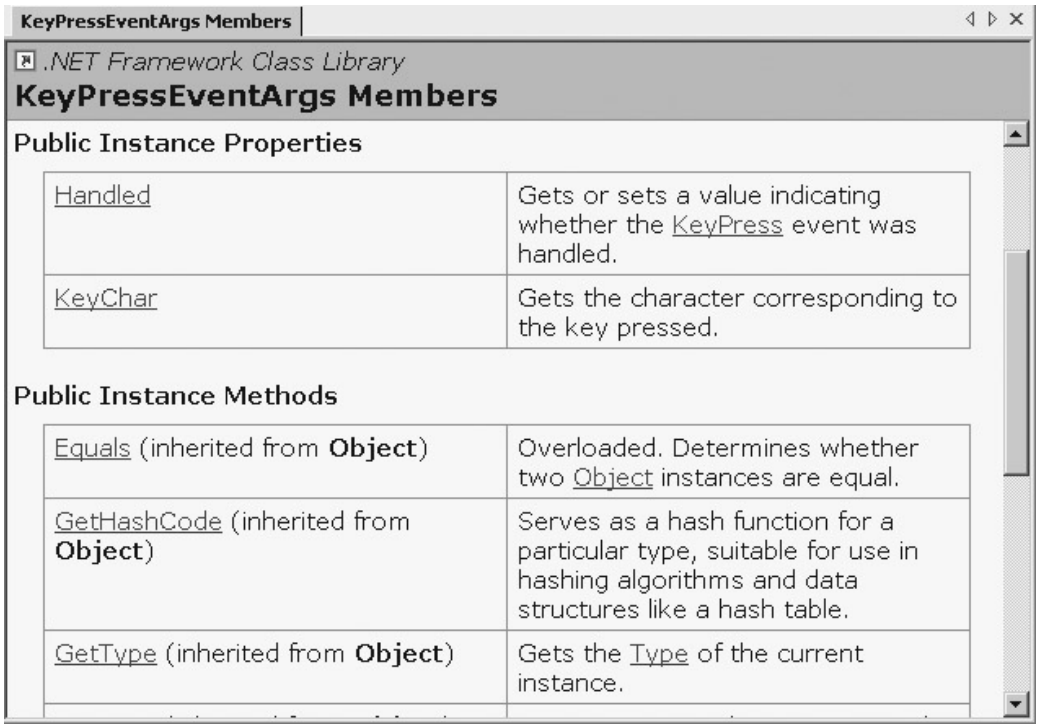

KeyPressEventArgs Members
.NET Framework Class Library
KeyPressEventArgs Members
Public Instance Properties
Handled
Gets or sets a value indicating whether the KeyPress event was handled.
KeyChar
Gets the character corresponding to the key pressed.
Public Instance Methods
Equals (inherited from Object)
Overloaded. Determines whether two Object instances are equal.
GetHashCode (inherited from Object)
Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures like a hash table.
GetType (inherited from Object)
Gets the Type of the current instance.

Figure 7.5: The KeyPressEventArgs object has properties and methods describing which key was pressed.

Trick Whenever you are trying to learn about a new kind of event handler, take a careful look at the parameters sent when that event is triggered. You can find these parameters either through the online help or by having the Designer create you an event of whatever type you're interested in. The parameters are often classes that provide all kinds of useful

information about the event.

I wanted to copy the value the user typed to lblPress. The code is simplistic:

```
private void KeyReader_KeyPress(object sender,
             System.Windows.Forms.KeyPressEventArgs e) {
      lblPress.Text =  "Press: " + Convert.ToString(e.KeyChar);
    }  // End KeyPress
```

e.KeyChar refers to one of the properties of the KeyPressEventArgs object. The KeyChar property returns the key the user just tried to type. KeyPress events are good at retrieving the same kinds of values you might see in a text box (although one at a time). You cannot use the KeyPress event to look for control keys, arrow keys, or function keys. These keys simply do not trigger the KeyPress event, so you can't use a KeyPress() method to look for them.

In the Real World

After all the advice about long, descriptive variable names, you might wonder why a variable as important as the KeyPressEventArgs has a name like e. I can't be sure what was happening in Redmond, Washington, when the C# developers were planning their language, but this is one of several areas where C# looks suspiciously like another language, called Java. Most event handlers in Java pass an event object named e (for *event*). In C#, it isn't called an *event* object, but the variable is used in exactly the same way and is still named e. Certainly, the key people involved in the development of C# were aware of Java, and most were probably fluent in the language. It's not surprising that many of Java's best ideas found their way into C#—but maybe I'm being cynical. Maybe e is shorthand for KeyPressEventArgs.

Trick The KeyPress event is great when you want to watch what the user is typing. For example, you might have a text box where you want the user to enter only numbers. You can use the KeyPress() method to check each key as it is pressed and ensure that it is a number. The Backspace key, arrow keys, Alt key, Ctrl key, and Shift key should operate normally in these situations, without triggering the KeyPress event.

The e.KeyChar property returns a variable of the char type, described briefly in Chapter 2, "Branching and Operators: The Math Game." A char contains only one letter. chars aren't very useful on their own, so usually you convert them to strings before you do anything else with them. I did so in this case, with your old friend the convert object. I then added the value from the key press to the appropriate label's text property.

Coding the KeyDown Event

Having three events for handling keyboard input might seem redundant. However, KeyDown and KeyUp act differently than KeyPressed and are used in different circumstances. KeyDown is activated whenever a key is held down. If the key is held down for a long time, the event is called continuously. The KeyUp event is triggered whenever the user releases the key. The KeyDown event is very useful for game programming because it can be triggered continuously and can trap for any key on the keyboard, including the function keys and arrow keys. In addition to the differences in timing, the KeyDown and KeyUp keys are different from KeyPress because they are more closely mapped to the hardware than the KeyPress event. The KeyPress event is used to determine what the user intended to type. KeyUp and KeyDown can determine exactly what key is being held down at any one time. This is a subtle but powerful difference. Look at the template code provided for the KeyDown event:

```
private void KeyReader_KeyDown(object sender,
        System.Windows.Forms.KeyEventArgs e) {
} // end KeyDown
```

You might miss something important (I did, on the first attempt). KeyDown looks much like KeyPressed, and both have a parameter named e. The es are not exactly the same. The e in KeyDown is an instance of the KeyEventArgs object, which is a different beast than the KeyPressedEventArgs object. As you can see from Figure 7.6, the KeyEventArgs object is the more powerful of the two objects and sports interesting properties.

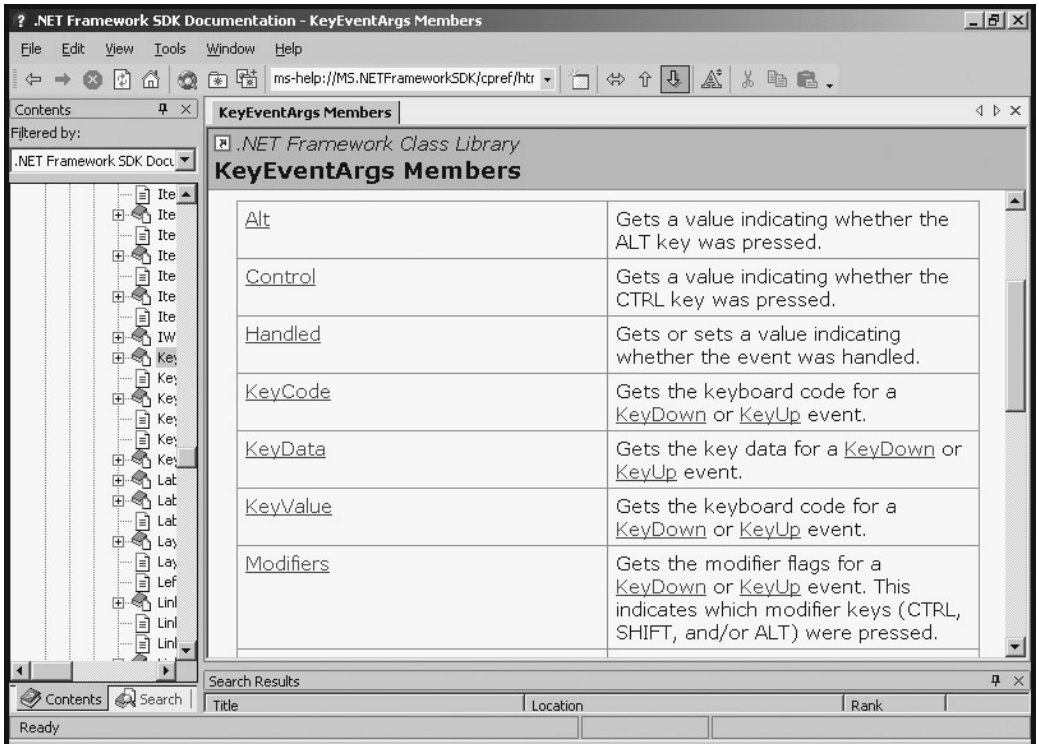

.NET Framework SDK Documentation - KeyEventArgs Members
File Edit View Tools Window Help
ms-help://MS.NETFrameworkSDK/cpref/htr
Contents
Filtered by:
.NET Framework SDK Docu
KeyEventArgs Members
.NET Framework Class Library
KeyEventArgs Members
Alt
Gets a value indicating whether the ALT key was pressed.
Control
Gets a value indicating whether the CTRL key was pressed.
Handled
Gets or sets a value indicating whether the event was handled.
KeyCode
Gets the keyboard code for a KeyDown or KeyUp event.
KeyData
Gets the key data for a KeyDown or KeyUp event.
KeyValue
Gets the keyboard code for a KeyDown or KeyUp event.
Modifiers
Gets the modifier flags for a KeyDown or KeyUp event. This indicates which modifier keys (CTRL, SHIFT, and/or ALT) were pressed.
Search Results
Title
Location
Rank
Contents
Search
Ready

Figure 7.6: The KeyEventArgs object has a longer list of features than KeyPressedEvents.

The code in KeyDown is much like that in KeyPressed, but I took advantage of the more powerful event argument object in KeyDown. KeyEventArgs does not have a KeyChar property. Instead, it returns an integer code named KeyValue. This code describes which key on the keyboard was pressed and also stores, in a special binary structure, information about which *modifier* keys (the keys that are meant to be pressed with another key, such as Shift, Ctrl, and Alt) are also pressed down. Getting meaningful information out of the KeyValue property can be very ugly, but KeyEventArgs comes with some other, more friendly properties that make it much easier. KeyCode describes which key was pressed, and the Control, Alt, and Shift properties return true or false values describing whether the indicated modifier is currently pressed. Generally, you check the KeyCode property to determine which key was pressed.

In the Real World

The KeyPressed event handler is most useful in game situations, where it is usually used to trap for arrow keys and function keys. The KeyUp and KeyDown event handlers are useful when you want to check for control or alt sequences in your programs. I once wrote a paint program that allowed the user to change colors with a control key (control–B turned the pen blue, for example) My KeyUp event first checked whether the control key was currently being held down. If so, it checked whether KeyCode corresponded to the B key. If so, it turned the pen blue.

Determining Which Key Was Pressed

The KeyCode returns a special value that relates to the hardware location of whatever key the user pressed. For example, on many keyboards (but not all) the left arrow key might be mapped as key number 123. This information isn't that useful to a programmer, who wants to know whether the user hit the left arrow. .NET provides a special set of values with names, called an *enumeration*, so that you don't have to remember which particular number is associated with each key. Instead, you find the appropriate name from the list of names that pops up when you try to complete the code. All the key codes are stored in an enumeration named Keys. The KeyCode() method returns a value of the Keys enumeration, and you can compare it to other values from the enumeration. For an example, look at the code I added to the KeyDown event for the Key Reader program:

```
private void KeyReader_KeyDown(object sender,
        System.Windows.Forms.KeyEventArgs e) {
      lblDown.Text = "Down: " +
                     Convert.ToString(e.KeyCode);
      if (e.KeyCode == Keys.ShiftKey){
        MessageBox.Show("That is one shifty character");
      } // end if
    } // end KeyDown
```

The first thing you might notice is that I sent the value of e.KeyCode to lblDown, much as I did in KeyDown. The if statement compares the value of e.KeyCode to the Shift keys, which are represented in the Keys enumeration as Keys.ShiftKey.

Animating Images

Animation is an important feature of graphical programs. Most computer animation involves one or more of three basic techniques: creating a scene mathematically on–the–fly, moving an object on the screen, and swapping between several images in the same space to create the illusion that the image is changing. In this chapter you focus on the last two techniques. Figure 7.7 illustrates an example of the image–swapping form of animation. The Spin Globe program shows several images placed in succession in the same spot.

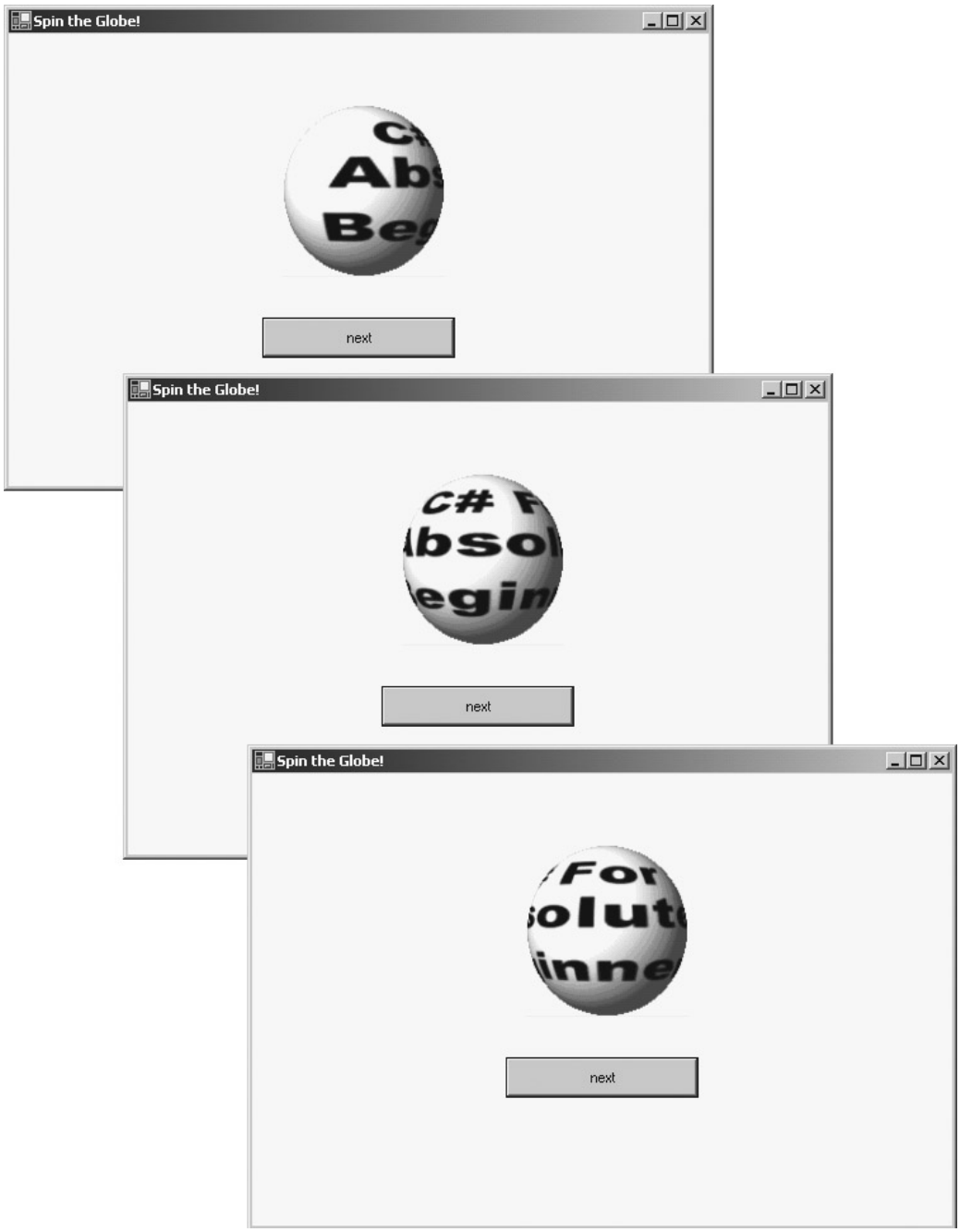


Figure 7.7: Each time the user clicks the Next button, the image changes. The result is the illusion of a spinning globe.

The Spin Globe program has only one picture box on it, but that picture box displays several values. If the user presses the button repeatedly, the ball appears to be spinning. The sequence repeats indefinitely.

Introducing the ImageList Control

The secret weapon of animation is an object named the ImageList control. The ImageList is a special object used to store a series of images offstage. It is not visible on the form, but you can copy images from the ImageList to a visible control, like the picture box in the Spin Globe program (refer to Figure 7.7). To use an ImageList, select it from the toolbox and drag it onto the form as you would any other control.

When you drag an ImageList object onto the screen, it doesn't stay there. Instead, it goes to a special "offstage area" below the normal Form Designer. You can see where the ImageList object is placed by looking at Figure 7.8.

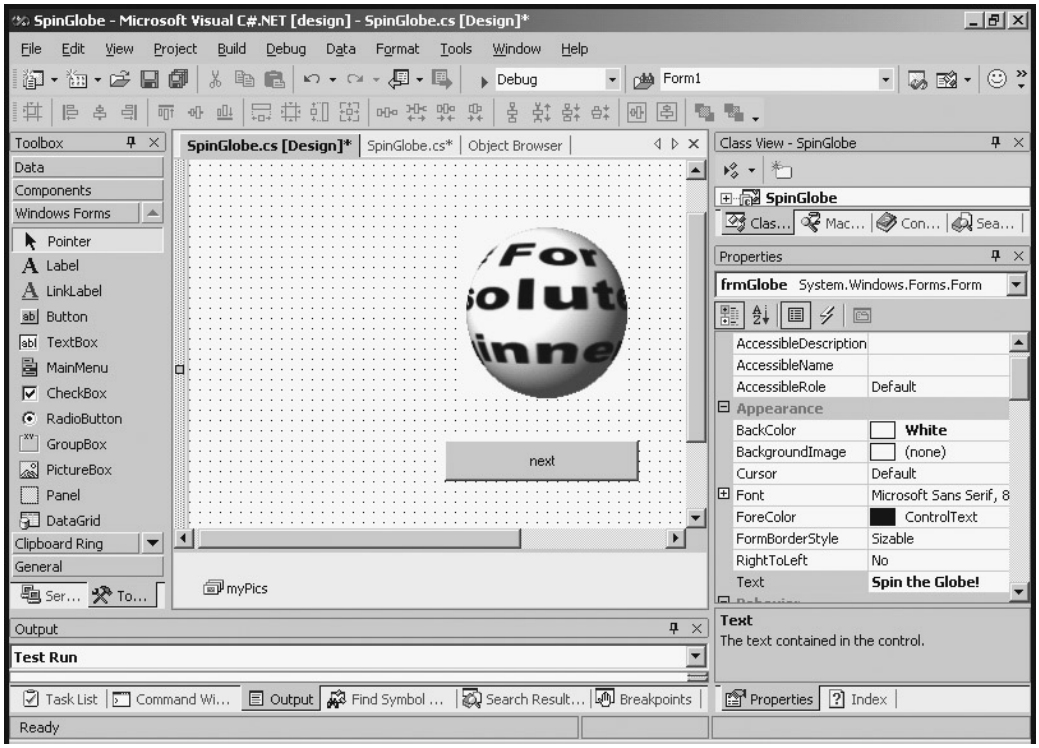


Figure 7.8: The ImageList control resides in a special place outside the main screen.
The special area below the form is used to indicate controls that are attached to the form but have no visual representation when the program is running. The ImageList control is one such control.

Setting Up an Image List

Although the ImageList control does not take up any space on the form, it still has properties, like any other control. I changed the name to myPics in exactly the same way you change the name of any other object, through the Properties window. The Images property is important because it enables you to load several images into the ImageBox control's memory, where they are available for later retrieval. When you activate the Images property, you see a special dialog like the one in Figure 7.9. This screen allows you to grab images to store in your programs.

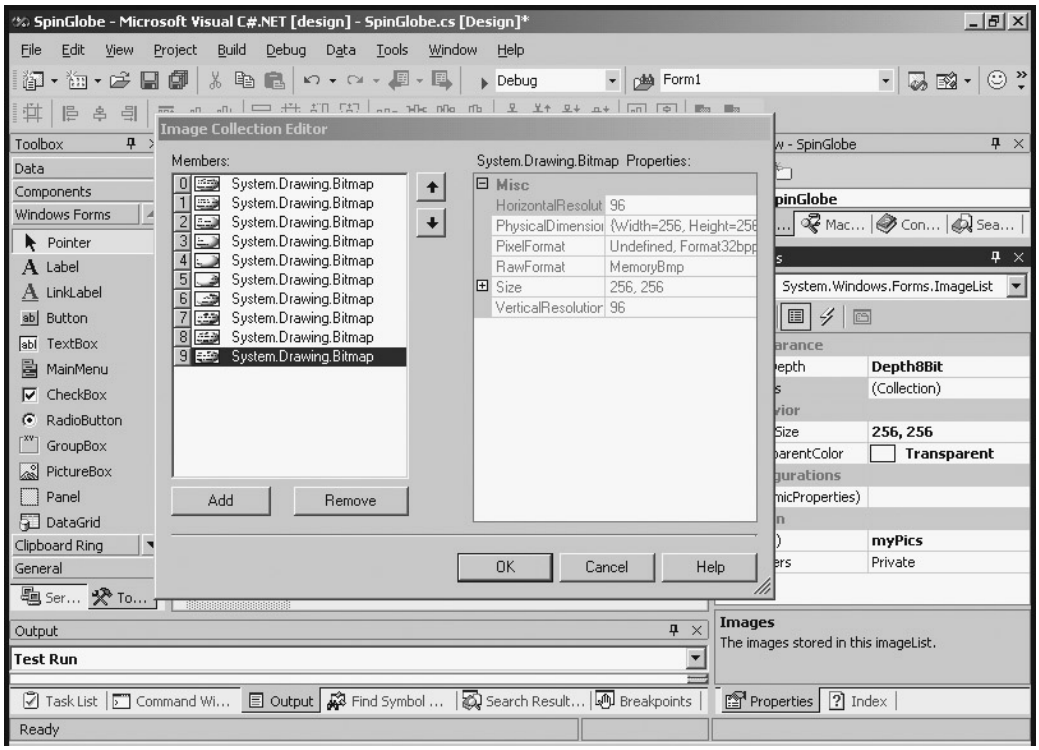


Figure 7.9: When you modify the Images property of an ImageList, the program provides a special editor where you can select each image from your file system.

Trap Be sure to set the ImageSize property whenever you use an ImageList control. The default size (16 by 16 pixels) is extremely small and causes your images to either show up very small or appear grainy (depending on the SizeMode of the picture box you place the image in). Generally, you want the ImageSize value set to about the same size as the images you'll be displaying on the screen. If you set the value too small, you do not preserve enough resolution. If you set the value too large, you can waste a great deal of memory. I usually use 256 by 256 as a starting size. This is a reasonably good compromise. You should set the image size on the image list before you start adding images to the list.

In the Real World

The ImageList control might not seem necessary, but it is very useful. When you store all the images you will need in an ImageList, the images are saved as part of your program. This makes your program much more portable because you don't have to worry about which drive the user has stored images on, for example. Images stored in an ImageList are loaded into the computer's memory when the program first loads, so the images can be displayed in very rapid succession. If the computer has to retrieve each image from the disk (the other main approach to this type of animation), there is a lag as each image is loaded from the disk. The ImageList control helps to solve all these problems.

Looking at the Image Collection

The ImageList control stores all the images into a special structure called a *collection*. The collection is basically a list of objects. Each image in the list has a number. You can use the Add button in the Image Collection Editor (the form shown if Figure 7.9 that pops up when you activate the Images property) to add a new image to the list. You can also use the Delete button to take an image out of the list and use the arrow keys to manipulate the order of images.

Trick You can get good image sequences for animation in many ways. If you are a talented artist, you yourself can generate them. If your artistic skills are not so good (mine aren't), you can often overcome innate lack of talent with time, technique, and good software. I used an open−source image program named *The Gimp* to generate the images for this program (and, in fact, every image and screen shot in this book). I used a special script named *Spinning Globe*, which automatically maps any image onto a globe and creates several frames perfect for animating. A copy of The Gimp is available on the CD−ROM that accompanies this book. It features many other very powerful animation scripts and a host of other features. Of course, you can also use any high−end image−editing program you want. I recommend that your image software have support for transparency, layers, and color manipulation tools. A person with even a modest drawing ability can do good work using such tools.

Displaying an Image from the Image List

Almost all the code for the Spin Globe program is in the click event of the button. I added one class−level variable named counter:

```
private int counter = 0;
```

The counter variable is used to determine which frame of the animation should be displayed. I declared it outside any methods but inside the class. I'll explain in a moment why I did that, but take a look at the code for the button click first:

```
private void btnNext_Click(object sender, System.EventArgs e) {
  counter++;
  if (counter >= 10){
    counter = 0;
  } // end if
  picGlobe.Image = myPics.Images[counter];
} // end btnNext_Click
```

Each time the button is clicked, the value of the counter variable is incremented by 1 with the counter++ statement. The value of counter relates to the indices of the images in the ImageList. If you refer to Figure 7.9, you'll see that the images are numbered from 0 to 9. I carefully set up counter so that it would count from 0 to 9 as well. It's important to notice that the counter variable was declared inside the class but not inside the method. If counter was declared inside the method, it would be reinitialized each time the button is clicked, and its value would never get beyond 1. Because counter was declared at the class level, it keeps its value while the class is running, and it can be accessed multiple times from the method without problems.

Trap If you've programmed in older (non .NET) versions of Visual Basic, you might remember the static keyword that was often used to refer to exactly this kind of variable. C# supports the static keyword as a variable modifier, but this does not work in the same way. It's better to declare a variable at the class level if you will need it in more than one method or if the same method will need to keep track of its value across multiple calls.

The critical part of the button_click code is the line that copies myPics.Images [counter] to the Image property of the picture box. It takes the element from the image list with the same index as counter and displays that image in the picture box.

Using a Timer to Automate Animation

Although the Spin Globe program is fun, it would be better if the animation could happen automatically, without the user's having to do anything. You can make this automatic animation happen by employing one new control to the Spin Globe program. The Auto Spin program shown in Figure 7.10 works much like the Spin Globe program but doesn't have any buttons at all. Instead, a new type of invisible control, named the *timer*, automates the swapping of images.

Figure 7.10: You can't see it here, but the image changes 10 times per second.

Introducing the Timer Control

The timer control is another control that is visible when you design a form but invisible to the end user. Like the ImageList, it appears in a special offstage area at the bottom of the form. Figure 7.11 shows my editor as I was building the Auto Spin program.

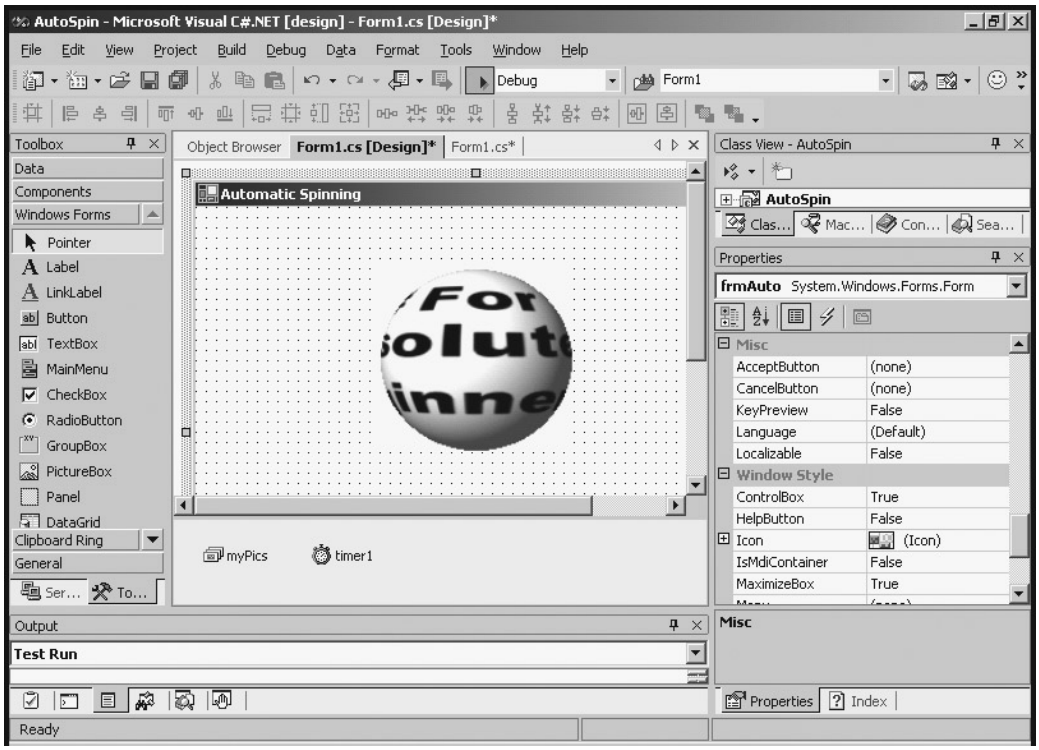


Figure 7.11: The Auto Spin program features two invisible controls and one picture box that is apparent to the user.

The code for the Auto Spin program is almost completely identical to that of Spin Globe. The only difference is that I replaced the button with a timer. The timer control is set to trigger an event at regular intervals. It has two primary properties. The Enabled property turns the timer on and off, and the Interval property describes how many milliseconds (1000ths of a second) between events. If you set the interval to 500, for example, the events occur every half–second.

Trick In game programming, the general rule of thumb is 10 frames per second, so you often see the timer interval set at 100, which generally causes the game's animations to run at that speed. However, this interval is only a guideline. Even if you set the interval to 1 millisecond, it's unlikely that the computer will be able to comply because there may be too many calculations for the processor to manage in the given interval. If you specify an interval the computer cannot manage, it will just go as fast as it can. An interval of 100 is more than enough for most simple games and animations and slow enough that it probably won't bog down the rest of the computer.

Configuring the Timer

To use a timer in your program, drag the timer control onto your form, set the Enabled property to true, and set the interval to whatever speed you want.

Trap Don't forget to set the timer's Enabled property to true. The default value is false, which means that the timer will not fire off at all and your carefully planned game or animation will just sit there doing nothing.

After you add a timer, you can add code to its only event—Tick. The Tick event occurs at the intervals you specified with the Interval property. If Interval is 250, any code in the Timer1_Tick() method will happen every 250 milliseconds, which is four times a second. For the Auto Spin program, I set the interval to 100 milliseconds. The code inside Timer1_Tick() is identical to the

code used in the Button_Press() method in the Spin Globe program:

```
private void timer1_Tick(object sender, System.EventArgs e) {
  counter++;
  if (counter >= 10){
    counter = 0;
  } // end if
  picGlobe.Image = myPics.Images[counter];
}
```

Adding Motion

Another critical type of animation involves moving objects around on the screen. This can be a simple matter when you understand how C# deals with locations on the screen. Figures 7.12 and 7.13 illustrate an object moving on the screen under user control.

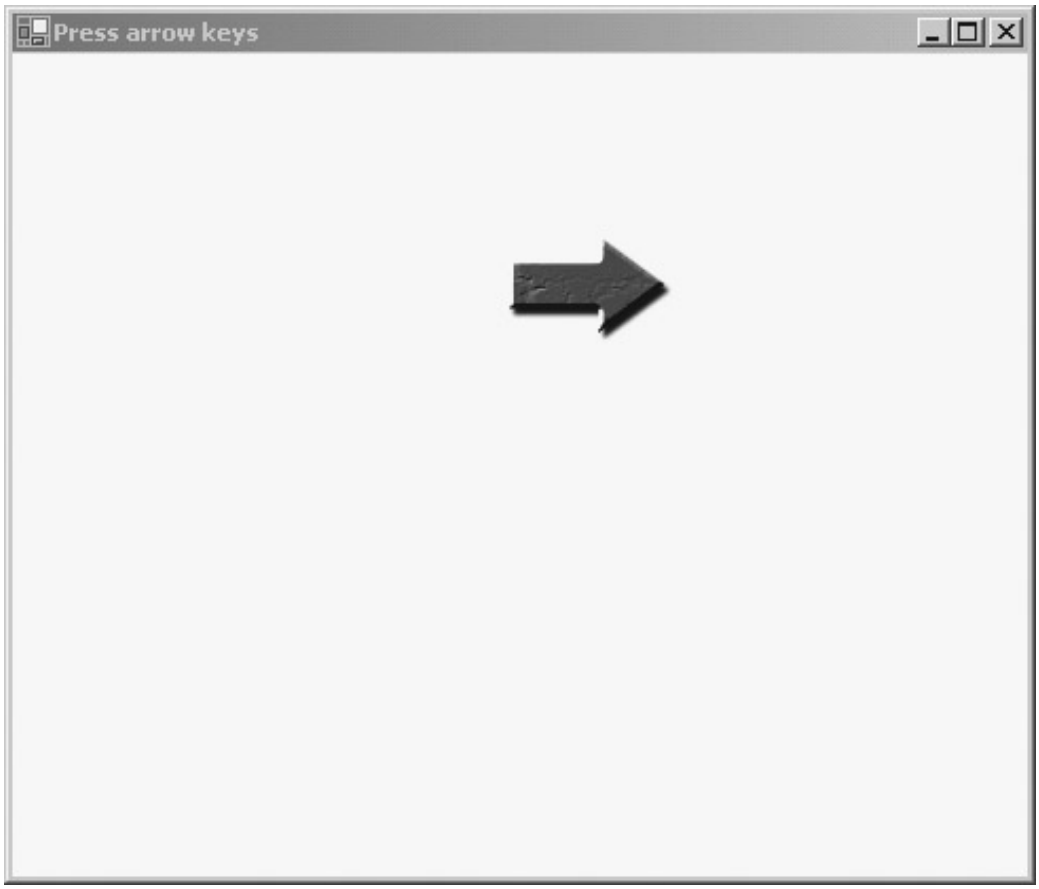


Figure 7.12: When the program starts, the arrow is moving to the right.

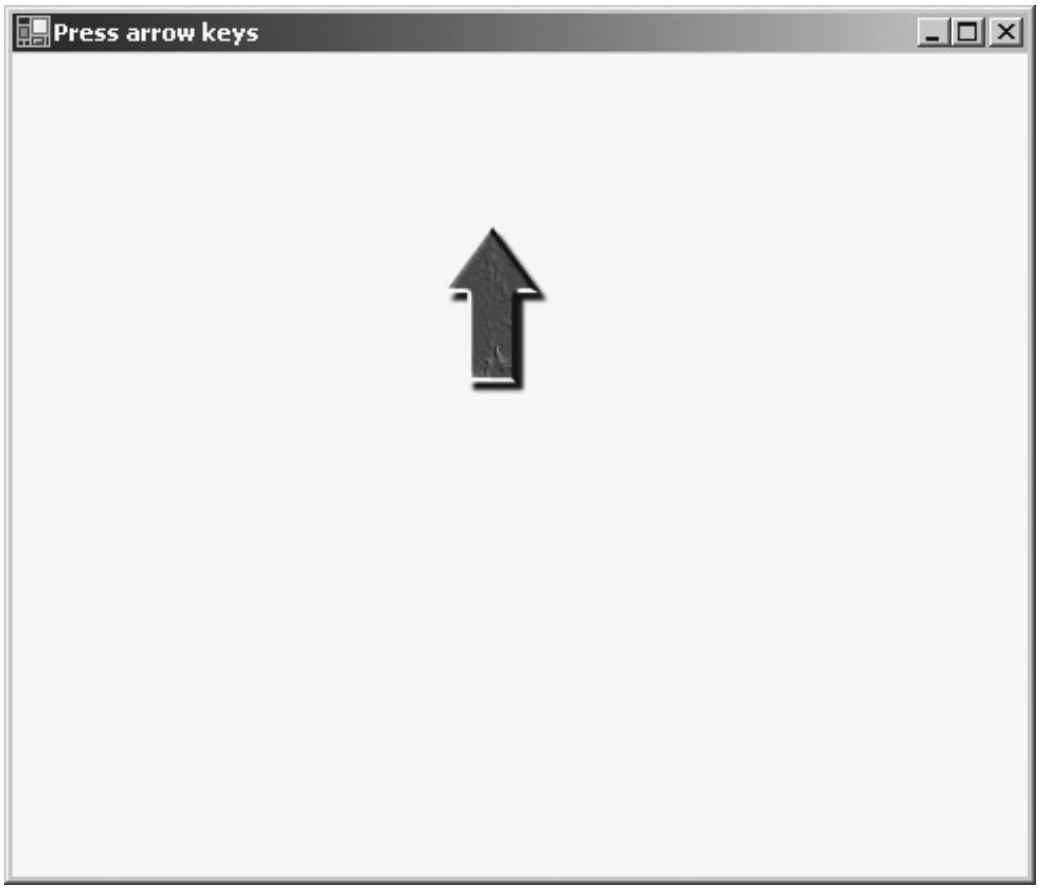


Figure 7.13: When the user presses an arrow key, the arrow image changes and moves in the indicated direction.

The easiest way to move an image in C# is to put the image in a picture box and move the picture box. This is easy to accomplish. The Mover program features one picture box named picArrow, an image list named myPics, and a timer left at the default name of timer1.

I declared two integer variables at the form level. The dX variable is meant to hold the difference in X (or delta X, if you want to sound like a rocket scientist). In other words, the dX variable determines how much the picture box will move in the X (that is, side–to–side) dimension. In computer graphics, an X value of 0 means the left side of the screen, and larger values move to the right. In a moment, you will add code that moves the picArrow picture box according to the value of dx. If dx is 0, the image does not move horizontally at all. If dx is negative, the image moves to the left, and if dx has a positive value, the picture box moves to the right. The dy variable controls the amount of motion in the Y axis, which determines whether the box moves up or down. Be careful, though, because in computer graphics, a Y value of 0 is at the top of the screen. As Y values get larger, you move *down* the screen.

Trap Because of the way computer screens work, the coordinate system in most programming languages differs from what you might remember from geometry class. The upper–left corner of the screen is (0,0). X values increase as you move to the right, and Y values increase as you move towards the *bottom* of the screen. It's very easy to become confused about this, especially with Y values. However, it's a relatively simple problem to spot and solve because you'll find your anvils (for example) floating up rather than dropping to the earth as proper anvils do. If your objects are moving up when they should be going down, or vice–versa, you've probably forgotten the upside–down way Y values work in the computer world. Fortunately, it's easy to fix.

Checking for Keyboard Input

The Mover program uses keyboard input to determine which direction the arrow should point and move. Here is the code to handle key presses:

```
private void theForm_KeyDown(object sender,
    System.Windows.Forms.KeyEventArgs e) {
  switch (e.KeyCode){
    case Keys.Right:
      picArrow.Image = myPics.Images[0];
      dx = 2;
      dy = 0;
      break;
    case Keys.Down:
      picArrow.Image = myPics.Images[1];
      dx = 0;
      dy = 2;
      break;
    case Keys.Left:
      picArrow.Image = myPics.Images[2];
      dx = -2;
      dy = 0;
      break;
    case Keys.Up:
      picArrow.Image = myPics.Images[3];
      dx = 0;
      dy = -2;
      break;
  } // end switch
} // end KeyDown
```

As you can see, the keyboard handler is essentially a switch statement based on which key was pressed (which is discovered by investigating e.KeyCode). This is a common pattern. Keyboard handlers very often follow this type of structure. The actual code for each key press is similar as well. The program replaces the Image property of picArrow (which is, as you recall, the visible picture box showing the arrow) with the appropriate image from the ImageList control (myPics). The program then changes the values of dx and dy to indicate how the image will move. Note that I have not actually moved the object yet. If I had put the movement code in this method, the arrow would move when the user presses a key, but only then. I want the arrow to keep moving in the direction of the last key press, so I need to put code somewhere else. (If you must know, that somewhere else is in a timer_tick event, which you'll see in a moment.)

Working with the Location Property

You can set and retrieve the location of any component by accessing that component's Location property. Location is (of course) an object. To be specific, it is an instance of the System.Drawing.Point class. The best way to move a component is to make a new Point object and then assign that point to the Location property of the component. The actual movement of the picArrow object happens in the Tick event of the Timer1 class. Here is the code:

```
private void timer1_Tick(object sender, System.EventArgs e) {
      int newX, newY;

      //change X value
      newX = picArrow.Location.X + dx;

      //check right boundary
      if (newX > this.Width){
```

```
      newX = 0;
    } // end if

    //check left boundary
    if (newX < 0){
      newX = this.Width - picArrow.Width;
    } // end if

    //change Y value
    newY = picArrow.Location.Y + dy;

    //check top
    if (newY < 0){
      newY = this.Height - picArrow.Height;
    } // end if

    //check bottom
    if (newY > this.Height){
      newY = 0;
    } // end if

    //create new point, assign to picArrow
    Point newLoc = new Point(newX, newY);
    picArrow.Location = newLoc;
  } // end timer tick
```

Getting New Values for X and Y

I started the method by creating integers named newX and newY. These integers will be used to build the point that will eventually be the new location of picArrow. I used the variables to simplify checking for boundaries. I started by copying the X location of picArrow into the dx variable. picArrow's Location property has an X property that provides this value. I then added dx to newX. The value of dx will be negative if the user wants to move to the left, positive if the user wants to go right, and 0 if the user does not want to move at all in the horizontal axis. If dx is negative, the value of newX will be smaller than it was the last time the timer ticked, and the box will move to the left when newX is applied to the picture box. Likewise, a positive value in dx will cause the box to move to the right, and a 0 value for dx will keep the box in the same position.

Checking for Boundaries

As I've said before, whenever you increment or decrement a variable, you should be sure to check for upper or lower limits. This is an especially important consideration in code attached to a timer, because that code is designed to run at frequent intervals. If something goes wrong in timer code, it will go wrong 10 times per second (at least, as the timer is set up in this program).

Wrapping the Arrow around the Screen

The next consideration is what should happen when the arrow moves off the screen. In this particular case, I decided to have the picture box wrap to the other side of the screen. If newX is less than 0, the picture box is moving off the left side of the screen, so I'll move it to the right side. However, it isn't always easy to tell how wide the screen is because your user can change the size of a window at any time. Fortunately, you have access to a special value named this. The this keyword refers to the class you are currently defining. In most Windows programming, this refers to the form on which your program is based. You can get the width of the form by looking at this.Width. Not surprisingly, the height is stored in this.Height. If the arrow goes past the right side of the form (this.Width), the program replaces it at the left side, which is always 0. Notice that the location of the picture box refers to the upper–left corner of the box. If you want to place the picture box so that its

right border is touching the right edge of the form (that is, it starts entirely visible along the right edge of the form), you place it at this.Width—picArrow.Width.

In the Real World

You can do essentially three things to an object when it moves off the screen. You can have it wrap around to the opposite side of the screen, as I did in the Mover program. This is a useful effect when you want to emulate limitless space. It's often used in space games. Sometimes you see map games that allow wrapping along the sides but not the top of the map, to simulate the effects of playing a game on a globe.

The second thing you can do is to have the object stop when it hits the edge of the screen. Do this by setting both dx and dy to 0 when the object encounters the edge of the screen. You can use this effect in a racing game, for example.

The third effect is to have the object appear to bounce off the screen. This is easily done by inverting the dx or dy value, as appropriate. For example, if an object hits the right border of the form, its dx must be positive, so subtract dx from 0 to get a negative value. If you like, you could also take into consideration the energy lost when an object bounces off something. If it hits the wall at dx of 4, set dx to –3 so that it moves back to the left but with less energy. You will see a version of this technique in the Crasher program later in this chapter.

Regardless of the technique you choose when hitting a boundary, using dx and dy variables greatly simplifies your coding of graphic objects.

Assigning the New Location to the Picture Box

Of course, the whole point of figuring out newX and newY is to determine where the picture box should go. Start by creating a new instance of the Point object, based on newX and newY:

```
Point newLoc = new Point(newX, newY);
```

Then assign that value to the Location property of picArrow:

```
picArrow.Location = newLoc;
```

Detecting Collisions between Objects

If you are moving objects around on the screen, sooner or later they are going to bump into each other. In arcade programming, the most important parts of the game are often related to these events. If you have a racing game, you need to know when the car passes the start line or hits a pedestrian (I mean, safety barrier—I don't write that kind of game). In a shooting game, you want to keep track of whether the missile hits the target. Game programmers have used *collision detection* algorithms for years to determine whether two objects overlap on the screen. There are several approaches to collision detection, but C# makes one form exceptionally easy to implement. I'll illustrate with the simple program featured in Figures 7.14 and 7.15.

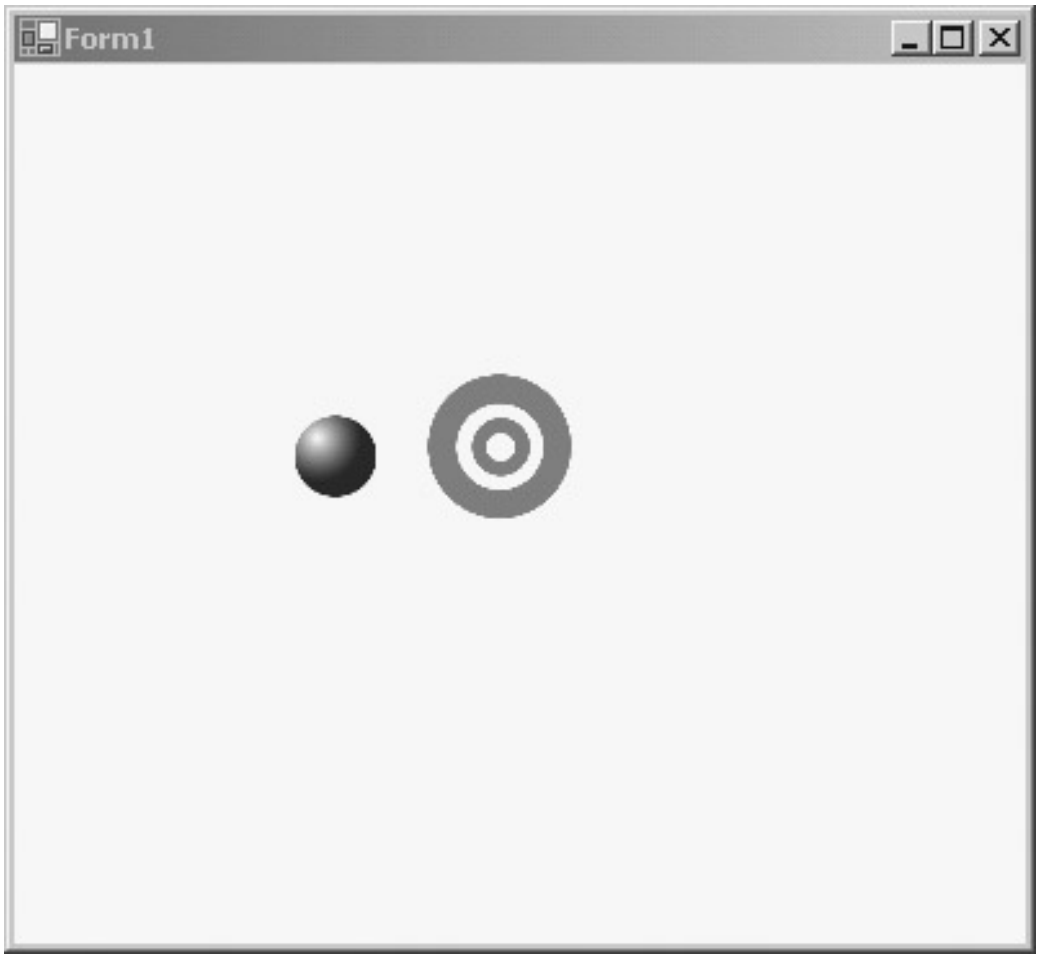


Figure 7.14: The ball is moving towards the target, and the background is white.

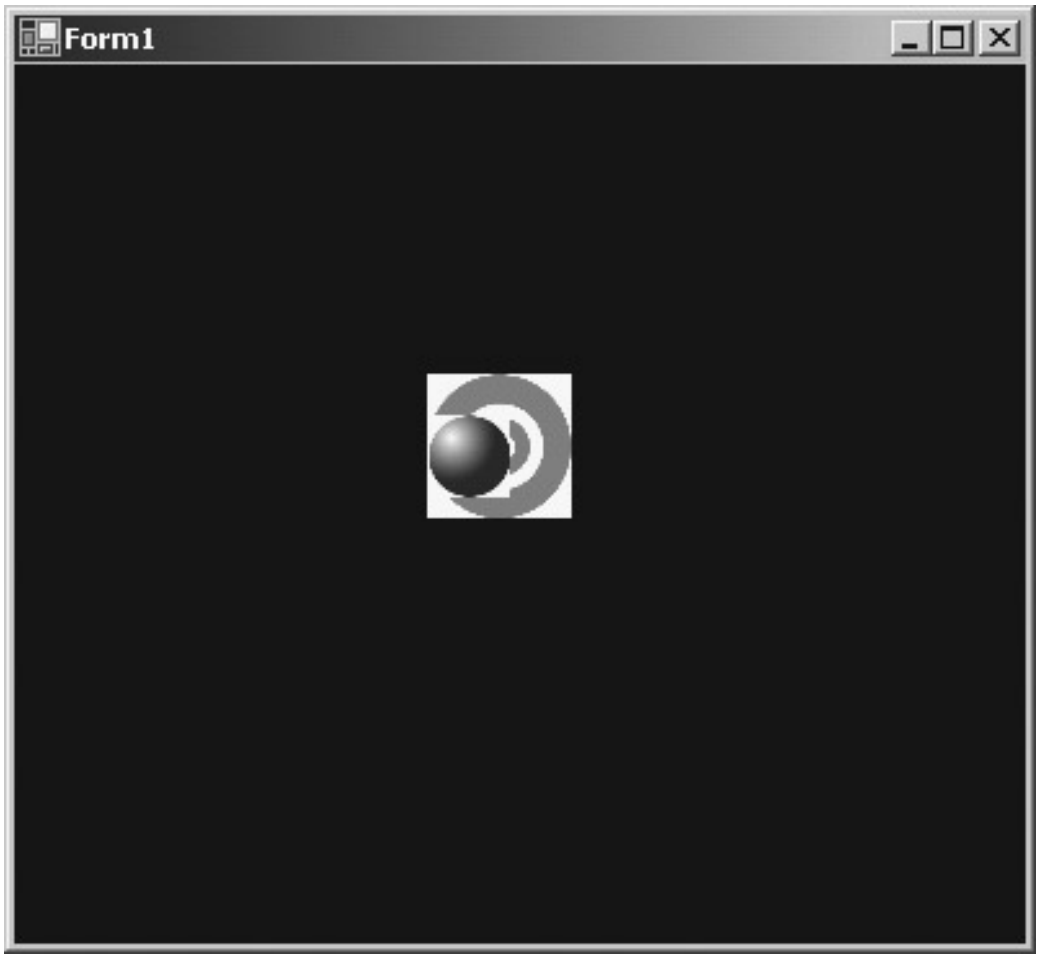


Figure 7.15: When the ball moves over the target, the form's background color changes to black.
The Crasher program is essentially two picture boxes, named picBall and picTarget. I set up picBall under the timer control so that it constantly moves back and forth on the screen. Whenever picBall collides with picTarget, the form's background changes color. This program is simple but illustrates how to test for intersections between objects.

Coding the Crasher Program

I created the Crasher form in the typical way. The only variable I added at the class level is dx, which is an integer value, initialized to positive 4. I didn't use dy at all in this program because the ball will not move vertically, only horizontally.

All the remaining code for the program goes in the timer's tick method:

```
private void timer1_Tick(object sender, System.EventArgs e) {
  int newX, newY;

  newX = picBall.Location.X + dx;
  newY = picBall.Location.Y;

  //check for borders
  if (newX > this.Width - picBall.Width){
    dx = - dx;
  } // end if

  if (newX < 0){
    dx = - dx;
  } // end if
```

```
  // look for collision
  if (picBall.Bounds.IntersectsWith(picTarget.Bounds)){
    this.BackColor = Color.Black;
  } else {
    this.BackColor = Color.White;
  } //end if

  picBall.Location = new Point(newX, newY);
} // end timer
```

Getting Values for newX and newY

I created the newX and newY variables just as in the Mover program. The newX variable is the X location of picBall + dx. This will effectively move the picture box dx pixels each time the timer ticks. The newY variable gets the Y location of picBall. Because I never add anything to newY, the ball will never move in the vertical axis. Still, I find it handy to have a newY value because this makes setting the location of the ball at the end of the tick method easier.

Bouncing the Ball off the Sides

As usual, when you move something, you should check for boundaries. In this case, I decided to bounce the ball whenever it hits a side. To reverse an object's direction, all you have to do is invert the value of dx. In other words, if dx is 4 and the ball hits the right side of the screen, set dx to –4. If dx is –4 and the ball hits the left side of the screen, set dx to 4. In either case, you can set dx to its inverse value by assigning its negative value. I did this in the code by setting dx = –dx.

Checking for Collisions

The actual collision is relatively simple to check for if you let the .NET objects do all the work. You yourself can come up with a formula to determine whether two objects collide. If you search around, though, you're likely to find that you have access to objects that will do this work for you.

Extracting a Rectangle from a Component

Each component is automatically given a special property named a Bound. The Bound is an instance of the Rectangle class describing where that object is on the screen. The Rectangle class has an IntersectsWith() method that is used to determine whether one rectangle intersects with another rectangle. When you know all that, you can easily write the code that determines whether the rectangles collided:

```
if (picBall.Bounds.IntersectsWith(picTarget.Bounds)){
```

picBall has a Bounds property, which is a Rectangle. All Rectangles have the IntersectsWith() method, which can accept another Rectangle. I sent the Bounds property of picTarget as a parameter to the IntersectsWith() method. The result of all this is a Boolean value just perfect for an if statement. If the rectangles intersect, I change the form's background to black. If not, I change the background to white.

Getting More from the MessageBox Object

By now, you've learned nearly everything you need in order to write the Lunar Lander game. However, that program has one, seemingly little, feature that is very useful and bears explanation. When the game is over, the user gets a message box that asks whether the user wants to continue. Such a message box requires more than one button. All the other message boxes you have seen have only one button. The two–button version of the message box uses the same MessageBox class you've already used, but it works slightly differently.

In the Real World

It is very reasonable for you to wonder how you're supposed to know that picBall has a Bounds property, that Bounds is a Rectangle, and so on. You *aren't* supposed to know it! The .NET environment is far too comprehensive for any one programmer to understand fully. It's more important that you know how to think about the environment and how to look up what you need than to have all the details socked away in your head.

I solved the collision detection problem by thinking through the STAIR process. I knew that I had two picture boxes, and I wanted to know whether there is a method that determines whether they intersect. I was surprised to find out that neither the PictureBox class nor any of its ancestors has any sort of collision method. I then went searching through the documentation for some type of object that *does* have the capability to check for intersections. I discovered that the Rectangle class has this capacity. After that, all I needed to do was figure out how to get a Rectangle related to a particular component, and I found the Bounds property. There are certainly other ways to solve this problem, but this one works. Programming is not about knowing how to do things. It's about knowing how to *figure out* how to do things.

Introducing the MsgDemo Program

I created a small program to illustrate how you can squeeze much more information out of the MessageBox class. Figures 7.16 through 7.18 show the MsgDemo program.

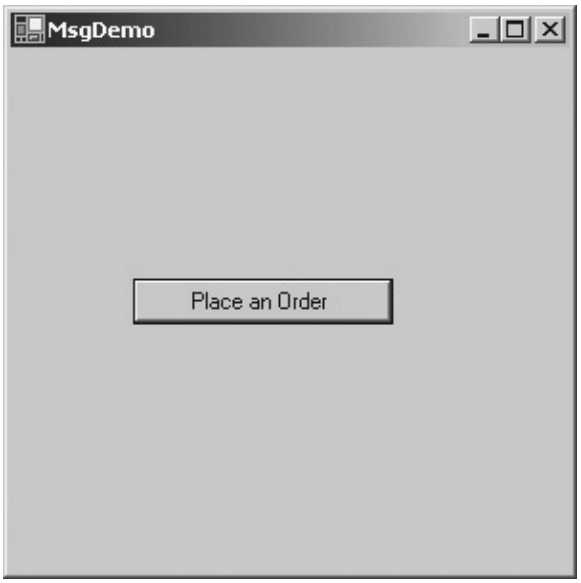


Figure 7.16: Nothing happens until the user clicks the button.

Figure 7.17: Notice the question icon, the two buttons, and the labels on the buttons.

Figure 7.18: Apparently, there is a mechanism for reading which button was clicked.

Message boxes seem to be extremely simple but can be very tricky to program. Part of the problem is the need for consistency. It's usually a good idea for your programs to use the same kinds of conventions your users have seen many times. Therefore, using the built–in dialogs is often better than building your own. Making basic modifications to the MessageBox class is easy but takes planning and digging around in documentation to find exactly what you want.

I wrote the MsgDemo specifically so that all the relevant code would go in the click event of the sole button. Here's the code:

```
private void btnYesNo_Click(object sender, System.EventArgs e) {
  DialogResult reply;
  reply = MessageBox.Show(
    "Do you want fries with that?",
    "Yes or No Demo",
    MessageBoxButtons.YesNo,
    MessageBoxIcon.Question);
  if (reply == DialogResult.Yes){
    MessageBox.Show("Here are your fries");
  } else {
    MessageBox.Show("No fries");
```

```
  } // end if
} // end btnYesNo
```

I started by creating an instance of the DialogResult class. When I investigated the Show() method of the MessageBox class, I discovered that it returned an instance of the DialogResult class. The reply variable is intended to catch whatever result comes back after the message box communicates with the user.

The Show() method of the MessageBox class is heavily overloaded. It has 12 variations! I used one with four parameters because it looked as though this version would provide all the details I needed.

In this four–parameter version of the show command, the first parameter is the message you want to ask the user. The next parameter is the title of the box that appears with the message inside it. (Users almost always overlook this title, but being able to change it is still nice.) The third parameter is a special value that determines which buttons to show, and the fourth parameter is another special value that describes the icon to display. Both these last values are enumerations.

An *enumeration*, as you may recall from earlier in this chapter, is a predefined list of values. The color constants are an example of an enumeration. In fact, any property that displays a drop–down menu in the Form Designer is likely associated with an enumeration. The MessageBoxButtons enumeration contains a list of all the possible button combinations. (Look in the online documentation to see all the choices.) Likewise, the MessageBoxIcons enumeration contains a list of all the possible icons you can place in a message box. When you understand how they work, you can usually use the syntax completion feature of the IDE to figure out which buttons and icons you want, without having to consult the online help each time.

Retrieving Values from the MessageBox

If you have more than one button on a message box, it is important to determine which button the user pressed. The DialogResults object is the key to figuring out the user's response. DialogResults has its own enumeration built in, describing all the various buttons that could have been pressed by the user. To figure out which button was pressed, simply compare the reply variable to the possible options in the DialogResults enumeration.

Coding the Lunar Lander

Building the Lunar Lander game simply requires putting together all the various elements in a new form. The program runs under a timer's control. All user interaction happens through keyboard input, and the animations use images copied from an image list.

The Visual Design

I started by sketching out the visual design and the overall plan for the game. The visual interface is very simplistic. I wanted one ship, named picLander, and one landing platform, named picPlatform. Additionally, I added a series of labels to communicate various game variables to the user. Each of these labels is named to correspond to a specific variable. I'll describe them in more detail in the ShowStats() method because I didn't add them to the interface until I wrote that method.

In addition to the visual elements, I added a couple invisible controls. A timer control handles all the interval–based elements (which take up the bulk of the code), and I used an image list to support the variations of the lander spouting flames in different directions. The image list includes four

images of the lander:

1. No flames
2. Flames on bottom
3. Flames on left
4. Flames on right

The Designer–Generated Code

Generally, I have not shown you the code generated by the Designer, but I show it to you here because I added a few elements. I'll explain my modifications after the code listing.

```
using System;
using System.Drawing;
using System.Collections;
using System.ComponentModel;
using System.Windows.Forms;
using System.Data;

namespace Lander
{
  /// <summary>
  /// Basic Arcade Game
  /// Demonstrates simple animation and keyboard controls
  /// and use of timer.
  /// Andy Harris, 1/16/02
  /// </summary>

  public class theForm : System.Windows.Forms.Form
  {
    //my variables
    private double x, y;      //will show new position of lander
    private double dx, dy;    //difference in x and y
    private int fuel = 100;   //how much fuel is left
    private int ships = 3;    //number of ships player has
    private int score = 0;    //the player's current score

    //created by designer
    private System.Windows.Forms.Timer timer1;
    private System.Windows.Forms.PictureBox picPlatform;
    private System.Windows.Forms.Panel pnlScore;
    private System.Windows.Forms.Label lblDx;
    private System.Windows.Forms.Label lblDy;
    private System.Windows.Forms.Label lblShips;
    private System.Windows.Forms.Label lblFuel;
    private System.Windows.Forms.PictureBox picLander;
    private System.Windows.Forms.ImageList myPics;
    private System.Windows.Forms.Label lblScore;
    private System.ComponentModel.IContainer components;

    public theForm()
    {
      //
      // Required for Windows Form Designer support
      //
      InitializeComponent();
      //I added this call, to a method that starts up a round
      initGame();
    }
```

```
/// <summary>
/// Clean up any resources being used.
/// </summary>
protected override void Dispose( bool disposing )
{
  if( disposing )
  {
    if (components != null)
    {
      components.Dispose();
    }
  }
  base.Dispose( disposing );
}

#region Windows Form Designer generated code

/// <summary>
/// The main entry point for the application.
/// </summary>
[STAThread]
static void Main()
{
  Application.Run(new theForm());
}
```

The Designer–written code shows its usual lack of flair but is functional. Note that I hid the InitializeComponent() method call because you shouldn't generally mess with it if you are using the Designer. I added several small but important things to the first chunk of code. I included a number of class–level variables and made a small modification to the constructor.

Class–Level Variables

Variables defined at the class level can be regarded as the DNA of an object. You can learn a lot about how the object works by understanding these variables, especially if the program is well written and documented. If these conditions are met, you can see an overview of the entire program's structure by looking at the variables.

Several of the variables (x, y, dx, and dy) are used to position the lander on the screen. You might be surprised to see that all four of these variables are doubles, rather than the integers you've used throughout the chapter for positioning components. The Point and Location classes that form the basis of screen motion require integers, but when I was testing the game, I found that integers did not give me the fine–grained control that I wanted. In particular, the effects of gravity were too difficult to get right, within the limits of integers. I decided to do my calculations with doubles and then convert the values to integers when needed. (Don't panic. I'll explain this as it comes up in the code listing.)

The fuel, ships, and score variables are used in scorekeeping and to determine when the game is over. All are integers.

> **Trick** Notice the comments after all the variables. Documenting all your important variables in this way is an excellent habit to form. You'll find that the effort pays off when your program becomes complicated and you don't remember what each of your cryptic variable names stands for.

The Constructor

You might recall that a *constructor* is a special method that helps build an instance of a class. It always has the same name as the class, and is automatically called when the class is created. In this case, the constructor is a method named theForm(). The Designer automatically created this constructor and added a call to the InitializeComponent() method (which it also created automatically). However, I also wanted something to happen the first time the program started. I wanted to set up all the initial conditions for the game. The instructions for this are stored in the initGame() method, which you'll investigate shortly.

Trap Sometimes you might be intimidated by the warnings not to change code created by the Designer. Although you certainly should be careful about doing this, *you* are the programmer. There is nothing wrong with adding your own code to a method generated by the Designer. In fact, it's often necessary to do so. Still, you should avoid changing InitializeComponent unless you're willing to finish writing the program without the assistance of the Designer. Changing the contents of that particular method can make it impossible for the Designer to read your code.

The timer1_Tick() Method

As you've seen throughout this chapter, much of the action in an arcade–style game happens in the tick method of the timer. The Lander game reinforces this observation.

```
private void timer1_Tick(object sender, System.EventArgs e) {
      //code that should happen on every timer tick
(10 times/sec)

      //account for gravity
      dy+= .5;

      //increment score for being alive
      score += 100;

      //show ordinary (no flames) lander
      picLander.Image = myPics.Images[0];

      //call helper methods to handle details
      moveShip();
      checkLanding();
      showStats();
    } // end timer tick
```

This method does a great deal of work. All the main logic for the game flows through this method 10 times a second. However, many of the most critical elements are passed off to other methods.

First, I added .5 to dy to account for gravity. Each time the timer ticks, there will be a small force pulling the lander downwards. The exact amount for dy is a tricky thing to determine. This was the main reason I used doubles for the math. When dy had to be a whole number, gravity of 1 was just too powerful at 10 times per second, and 0 gave no gravity at all. One solution to the "heaviness" of a gravity of 1 is to lower the frame rate by changing the timer's interval. When I tried this, the animation seemed too choppy. I decided, instead, to work with double values and then convert back to integers when needed. Sometimes you have to think creatively to get the results you want.

Then next thing I did was add a value to the score simply because the user survived another tick of the clock. It's a long–standing tradition in arcade games never to add fewer than 100 points to the

user's score. I guess that it's the video version of grade inflation.

I then displayed the version of the lander with no flames in the picture box by copying the appropriate image (number 0) from the ImageList. I did this because no flames should be the default setting. I'll add the flames later if I discover that the user is pressing a key.

The last three lines of the tick() method call other custom methods. The timer has three more important jobs to do, but each is detailed enough to merit its own method. I gave the methods names indicating the main jobs that need to be done. Leaving the details of (for example) moving the lander, updating the score, and checking for collisions to other methods is a good idea because leaving all that code in the tick() method would make the method extremely complicated and difficult to read and fix. With the custom methods, it's very easy to see the main flow without getting bogged down in details. Of course, you need to worry about the details at some point, but they're easier to work with in isolation.

Trick Whenever one method starts to be more than one screen long, consider breaking it into multiple methods. This way, you can break the work into smaller chunks that are easier to manage.

The moveShip() Method

The moveShip() method handles all the movement of the lander on the screen. The code should look familiar.

```
private void moveShip(){
  //change x and check for boundaries
  x += dx;
  if (x > this.Width - picLander.Width){
    x = 0;
  } // end if
  if (x < 0){
    x = Convert.ToDouble(this.Width - picLander.Width);
  } // end if

  //change y and check for boundaries
  y += dy;
  if (y > this.Height - picLander.Height){
    y = 0;
  } // end if
  if (y < 0){
    y = Convert.ToDouble(this.Height - picLander.Height);
  } // end if

  //move picLander to new location
  picLander.Location = new Point(Convert.ToInt32(x),
Convert.ToInt32(y));

} // end moveShip
```

The process of modifying dx and dy is probably routine for you by now, but this routine has one twist. Because I'm working in double values, I can't simply copy the screen location to x when I want to move to the right or the bottom of the screen. The compiler complains about the conversion from integer to double. I simply used Convert.ToDouble() to get past this problem.

Likewise, when I was ready to place the lander in its new position, the values of x and y were doubles, but I needed ints. Again, the Convert class came to the rescue.

The checkLanding() Method

The interesting moments in the game occur when the lander gets close to the platform. When these two components are in proximity, it means either a crash or a landing. The checkLanding() method determines whether the lander is near the landing pad and, if so, whether it is a safe landing or a horrible crash:

```
private void checkLanding(){

    //get rectangles from the objects
    Rectangle rLander = picLander.Bounds;
    Rectangle rPlatform = picPlatform.Bounds;

    //look for an intersection
    if (rLander.IntersectsWith(rPlatform)){
      //it's either a crash or a landing

      //turn off the timer for a moment
      timer1.Enabled = false;

      if (Math.Abs(dx) < 1){
        //horizontal speed OK
        if (Math.Abs(dy) < 3){
          //vertical speed OK
          if (Math.Abs(rLander.Bottom - rPlatform.Top) < 3){
            //landing on top of platform
            MessageBox.Show("Good Landing!");
            fuel += 30;
            score += 10000;
          } else {
            // not on top of platform
            MessageBox.Show("You have to land on top.");
            killShip();
          } // end on top if
        } else {
          //dy too large
          MessageBox.Show("Too much vertical velocity!");
          killShip();
        } // end vertical if
      } else {
        //dx too large
        MessageBox.Show("too much horizontal velocity");
        killShip();
      } // end horiz if
      //reset the lander
      initGame();
    } // end if
  } // end checkLanding
```

This code might look long and complex at first, but it is not difficult. The hardest part of writing this method was figuring out what constitutes a safe landing. I determined that a safe landing occurs only when all the following criteria are true at the same time:

- The rectangles for the lander and platform intersect.
- The horizontal speed (dx) of the lander is close to 0.
- The vertical speed (dy) of the lander is smaller than 3.
- The lander is on top of the platform.

The easiest structure for working with this kind of problem (where you can proceed only when several conditions are met) is a series of if statements nested inside each other. This structure is named (not surprisingly) *nested* ifs. I decided to turn each of the criteria in the list into an if statement and nest them all inside each other. In other words, I started by checking to see whether the platforms intersect. If you look back at the code, you see that the first if statement checks whether the rectangles intersect. If that condition is true, *something* has happened. If all the other conditions are true, the player has landed successfully, but if not, there has been a crash. If the rectangles do not intersect, there is no point checking the other conditions.

In the Real World

The nested if structure is commonly used whenever a programmer needs to check for several conditions. In more traditional programming (the kind you're more likely to get paid for), you commonly use nested ifs whenever you want to do some sort of validation. For example, if you write a program that processes an application form, you probably won't allow the program to move on until you are sure that the user has entered data in all the fields and that all the data can be checked. Validation code usually uses the same nested if structure as the Lunar Lander game. However, there are not as many cool effects when the user doesn't submit a valid email address. (I've always been tempted to add explosions in that sort of situation, but so far, I've been good.)

Checking the Horizontal Speed

If the rectangles *do* intersect, another if statement checks what the horizontal speed (measured with the variable dx) is. I wanted to require that the ship be nearly motionless along the X axis because the legs of such a craft can't handle much sideways velocity. (Also, this rule makes the game more challenging!) I figured that a value between –1 and 1 for dx would be a good range. However, testing for a value greater than –1 and less than +1 is a little ugly. I decided to take advantage of the built–in absolute value method of the Math class, Math.Abs(). As you may recall fondly from math class, the absolute value of a number strips the sign off a number, so the absolute value of –1 is 1, and the absolute value of +1 is also 1. By using the Math.Abs() method on dx, I was able to determine a very small horizontal velocity with only one if statement.

Checking the Vertical Speed

The vertical speed is calculated much like the horizontal speed. You might be surprised that I still used the absolute value function here because the lander will always approach the platform from the top and will always have a positive dy value if a legal landing is possible. This is true, but I seriously thought at one time about allowing landings from the bottom (maybe you're attaching a balloon to a floating platform). This would greatly change the strategy of the game and allow for interesting piloting situations, so I left the absolute value in here, just in case. (Does that sound like a *dandy* end–of chapter exercise, or what?)

Ensuring That the Lander Is on Top of the Platform

The last requirement for a healthy landing is that the lander must touch the top of the platform (at least for now). This turned out to be an easy thing to check. I just used properties of the picLander and picPlatform picture boxes to compare the bottom of picLander and the top of picPlatform. I used the absolute value method again to ensure that the bottom of the lander is within 3 pixels of the top of the platform.

Managing a Successful Landing

If the player passes all four landing tests, the program sends a congratulatory message, loads up more fuel, and adds significantly to the player's score. If the player fails to pass any of the landing criteria, but the rectangles intersected, the program responds with an appropriate message and calls the killShip() method, which handles the details of lander destruction.

Handling User Crashes

If the rectangles intersected (regardless of the rest of the consequences), the method calls the initGame() method to reset the positions of the lander and platform.

The theForm_KeyDown() Method

The player interacts with the program through keyboard commands. The up arrow fires thrusters that slow the effects of gravity and can eventually push the lander upwards. The left and right buttons fire side–pointing thrusters that allow side–to–side mobility. The keyboard handling routine is familiar if you investigated the Mover program earlier in this chapter:

```
private void theForm_KeyDown(object sender,
      System.Windows.Forms.KeyEventArgs e) {
      //executes whenever user presses a key

      //spend some fuel
      fuel--;

      //check to see if user is out of gas
      if (fuel < 0) {
        timer1.Enabled = false;
        MessageBox.Show("Out of Fuel!!");
        fuel += 20;
        killShip();
        initGame();
      } // end if

      //look for arrow keys
      switch (e.KeyData) {
        case Keys.Up:
          picLander.Image = myPics.Images[1];
          dy-= 2;
          break;
        case Keys.Left:
          picLander.Image = myPics.Images[2];
          dx++;
          break;
        case Keys.Right:
          picLander.Image = myPics.Images[3];
          dx--;
          break;
        default:
          //do nothing
          break;
      } // end switch
    }  // end keyDown
```

Every time the user presses a key (even non–arrow keys!), the program uses up one unit of fuel. As I've said many times, if you increment or decrement a variable, you should test for boundary conditions. Because I'm decrementing the amount of fuel, checking for an empty gas tank is

sensible. If the user runs out of gas, I disable the timer temporarily to stop the game flow and then display a message to the user so that he or she knows why the game stopped. I then call the killShip() method to take a ship out of the inventory, and I call the initGame() method to reset the speed and position of the lander and platform.

After dealing with the fuel situation, my attention turns to the actual key presses. Because I'm concerned with arrow keys here, I use the KeyDown() method and concentrate on e.KeyData. Depending on which key was pressed, I copy the appropriate image from the ImageList and set dx and dy to achieve the appropriate motion later when the timer ticks. Notice that I added a default condition to handle keystrokes other than arrow keys. If I were a nice guy, I would have used the default condition to add back the fuel value. Then, if the user accidentally hits a key, it would not cost precious fuel. However, as a game programmer, you can be mean if you want (cue maniacal laughter).

The showStats() Method

The showStats() method is called every time the timer ticks. Its job is to update the labels that display statistics, such as the score and the ships remaining to the user. This code is quite simple:

```
public void showStats(){
  //display the statistics
  lblDx.Text = "dx: " + dx;
  lblDy.Text = "dy: " + dy;
  lblFuel.Text = "fuel: " + fuel;
  lblShips.Text = "ships: " + ships;
  lblScore.Text = "score: " + score;
} // end showStats
```

A few assignment statements are all that is required. However, if you don't provide adequate information to the user, your game will not be successful. Also, because updating the score happens often, it's nice to have the code stored in a procedure.

The killShip() Method

The killShip() method is meant to be called whenever the user has lost a ship because of crashing or running out of fuel:

```
public void killShip(){
  //kill off a ship, check for end of game
  DialogResult answer;
  ships--;
  if (ships <= 0){
    //game is over
    answer = MessageBox.Show("Play Again?",
"Game Over",MessageBoxButtons.YesNo);
    if (answer == DialogResult.Yes){
      ships = 3;
      fuel = 100;
      initGame();
    } else {
      Application.Exit();
    } // end if
  } // end 'no ships' if
} // end killShip
```

The act of killing off the player's ship takes only one line of code. The real meat of the killShip() method is the part that checks whether the game is over. If the player is out of ships, the method asks the user whether he or she wants to play again, using a yes/no message box. If the player does want to play again, the number of ships is reset, and the initGame() method resets the speed and position of the lander and platform. If the user does not want to play again, the program exits completely with the call to Application.Exit().

The initGame() Method

The initGame() method is a real workhorse. It has a simple job but is called from several places in the program. The purpose of initGame() is to randomly set the location of the lander and platform and randomly choose the speed of the lander:

```
public void initGame(){
  // re-initializes the game
  Random roller = new Random();
  int platX, platY;

  //find random dx, dy values for lander
  dx = Convert.ToDouble(roller.Next(5) - 2);
  dy = Convert.ToDouble(roller.Next(5) - 2);

  //place lander randomly on form
  x = Convert.ToDouble(roller.Next(this.Width));
  y = Convert.ToDouble(roller.Next(this.Height));

  //place platform randomly on form
(but make sure it appears)
  platX = roller.Next(this.Width - picPlatform.Width);
  platY = roller.Next(this.Height- picPlatform.Height);
  picPlatform.Location = new Point(platX,platY);

  //turn on timer
  timer1.Enabled = true;

} // end initGame
```

With all the randomization that happens in the initGame() method, you won't be surprised to find an instance of the Random class defined in the method.

Choosing New dx and dy Values

I wanted dx and dy to be somewhere between 2 and –2, which is not directly possible with the Random class. However, I asked for an integer value between 0 and 4 (remember, the 5 refers to the *number* of possible responses, not the largest possible response), and I subtracted 2 from this number. This will give results evenly spaced between 2 and –2. However, the results are integers, and dx and dy are double variables, so I used the trusty convert object to get doubles where I wanted them.

Placing the Lander on the Form

Finding a legal place for the lander is relatively easy. The new X location should be somewhere between 0 and the width of the form, and the new Y should be between 0 and the height of the form. Again, it was necessary to convert to doubles because I chose to implement x and y as double values.

Placing the Platform on the Form

In my first attempt, I simply copied the lander form over to make the platform position code, but when I started play–testing, I discovered a problem. Every once in a while, the lander's position is mainly off the screen, so it is impossible to land on. Players don't mind being challenged, but they'll get grumpy if you make it *impossible* to succeed. I had to modify the code slightly to guarantee that the landing platform would be visible on the screen. You simply subtract the width and height of the platform from the screen width and height, and it becomes impossible for the random number generator to create a position that will cause the platform to be invisible.

Summary

In this chapter you learned how to build an arcade game, but the experience led you through several other important programming concepts as well. You learned how to add multiple images to an ImageList control. You learned how to read information directly from the keyboard in a couple different ways. You learned how to move objects around on the screen and what to do when they reach a screen boundary. You learned how to detect collisions between objects, and you created message boxes that can return values. In the next chapter you'll learn more about groups of objects and the fun things you can do with them.

Challenges

- Modify the difficulty level of the Lunar Lander game. There are several ways you could tweak the code. Perhaps you could change gravity by modifying the change in dy during each tick of the timer. You could also adjust how much dx and dy change during each key press or how fast time progresses by modifying the timer's interval. Another easy change would be to modify the size of the landing pad or the lander.
- Allow the lander to land on the bottom of the platform. This opens several new opportunities for changing the theme of the game. Maybe you are underwater in a buoyant craft and have to expend energy to go down, or maybe you can land from any angle because the platforms are somehow suspended in space.
- Have the platform move. The platform could start with no motion but then move slowly after the user has some success. Have the platform speed up as the user is more successful. Then add random erratic motion to the platform.
- Add obstacles to the game, perhaps having bits of floating debris the user must avoid. Similarly, you could add floating gas tanks for extra fuel, or other "powerups" that can make any landing safe, or provide some other short–term benefit.
- Add support for a second player. Add another lander, and read some other keys for the second player's controls. Award points only to the first player who lands in each round.
- Write a similar game based on the same ideas. You should be able to write a variant of the classic arcade game Joust with the skills you've learned in this chapter. In essence, that game places the user on an ostrich with a lance. You have to flap the ostrich to move vertically, and you have to spear opponents from above.

Chapter 8: Arrays: The Soccer Game

By now, you know how to control the logical flow of a program. Although this logical control is important, experienced programmers know that understanding how information is stored in your programs can be even more critical. By carefully designing your data schemes, you make your programs easier to read and maintain. Now you will find out how arrays are used to store lists of information. In this chapter you will learn how to

- Create and use single–dimension arrays.
- Use two–dimensional arrays to create tables.
- Work with the various methods and properties of the array object.
- Use the foreach loop with arrays.
- Write custom event handlers.
- Build visual components without the Form Designer.
- Register event listeners without using the Form Designer.
- Create arrays of visual controls.

The Soccer Game

The final project for this chapter is a simple game of soccer, as shown in Figure 8.1. The user is in control of a simplified team of five players: a goalie, fullback, halfback, wing, and center. At any time, only one of these players can have control of the ball. The user can click a player to attempt a pass to that player or can take a shot on the goal. The likelihood of a pass's being successful depends on which player has the ball and to whom he is kicking. For example, the goalie will more likely pass successfully to the fullback than to the center, who is usually lurking all the way down the field, because it’s almost impossible for the goalie to successfully shoot on the opposing goal. If the user is unsuccessful in a pass or shot attempt, the opposing team gets the ball and earns the chance to score. The game plays for one minute.

Figure 8.1: The player within the square outline has the ball. He can either pass to another player or make a shot on the goal.

Introducing Arrays

Although you can't tell by looking at the Soccer game from the outside, the game relies heavily on lists of information. Lists are so useful that every major programming language supports them in one way or another. Of course, programmers love fancy words and refuse to use the straightforward term *list*. They much prefer the more esoteric term *array*. An array is not complicated. It is simply a list of variables with the same name and the same type of data. If you've played golf (or even miniature golf), you've probably seen a scorecard that looks something like the one in Figure 8.2.

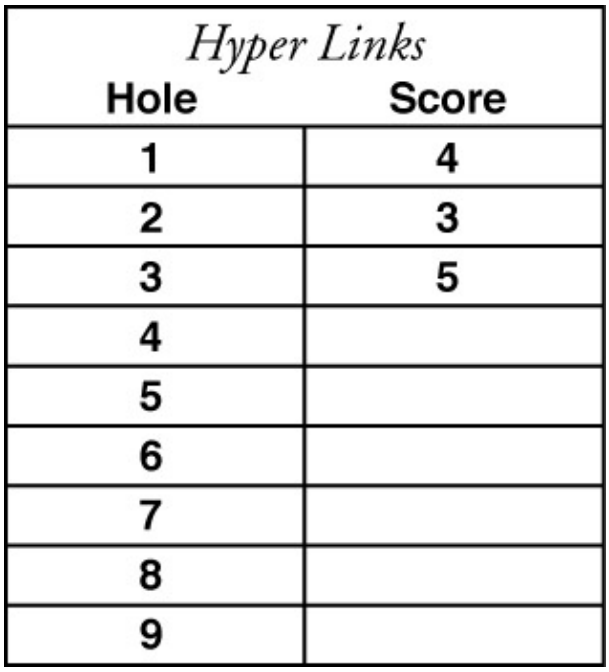


Figure 8.2: A golf scorecard is a good example of an array in everyday life.

A small golf course might have nine holes. These holes are usually numbered. Each golfer keeps track of how many strokes it takes to get the ball in the cup. The golfer might refer to the first score as the score for hole 1. The next score would be the score for hole 2, and so on. Programming languages usually refer to this kind of structure as an *array*. C# refers to an array with a bracket ([ ]) notation. If you wanted to make a golfing game in C# (hey, that's a great idea, but I've already thought up the game for this chapter), you would make an array of integers named score. You could assign the score for the first hole by writing score[1] = 3;. If you wanted to know the score for the fourth hole, you could write something like MessageBox.Show(score[4].ToString());.

Exploring the Counter Program

A sample program can clarify the concept of arrays, so take a look at the Counter program featured in Figures 8.3 and 8.4. The program counts to 4 and then repeats. The counting behavior itself is unremarkable. The more interesting problem is how to get the text values *One*, *Two*, and so on, in the right order. You guessed it. An array is the secret to this program's magic.

Figure 8.3: The label says *zero* before the user presses any buttons.

Figure 8.4: After the user presses the button, the value changes.

Creating an Array of Strings

The Counter program has only a few controls. The label holding the numeric text is named lblOutput, and the button is named btnNext. I added two variable declarations inside the class definition:

```
private int counter;
private string[] numbers = new String[5];
```

The counter variable is used to count numerically from 0 to 4 and also serves as an index in the array. Whenever you use an array, usually you also use an integer variable as an index.

The second statement declares numbers as an array of five strings. Making an array is different from making ordinary variables because an array is a type of object in C#. The term string[] indicates that you want to make an array of string objects, instead of an ordinary string variable. The = new String[5] part of the statement creates the array in memory and sets aside enough memory for five elements.

Trick If you don't know how many elements will be in your array, you can look up the more flexible ArrayList object. This object allows you to have an array with an indeterminate length. However, it requires much more work on the processor, so it can slow down your programs. Normal arrays are fine for most work.

Arrays are much like any other type of variable. They can be created at the class level or inside methods, and they can be declared with the public and private modifiers.

Referring to Elements in an Array

When you build an array you will also need to ensure that each element of the array has a starting value. This is called *initializing* an array. In the counter program, I initialized the array in the constructor:

```
public Form1()
{
  //
  // Required for Windows Form Designer support
  //
  InitializeComponent();

  numbers[0] = "zero";
  numbers[1] = "one";
  numbers[2] = "two";
  numbers[3] = "three";
  numbers[4] = "four";

} // end form1
```

The program assigns a different string value to each element of the array. Because numbers is defined as an array of strings, it can be referred to only with braces and an integer index, as shown in the preceding code. Also, notice that array indices start at 0, not at 1. The numbers array has five elements, but those elements are numbered 0 to 4, not 1 to 5, as you might expect.

Trap When you declare an array, you are determining the *number* of elements in the array, not the value of the largest index. If you create an array of 10 elements, the last one is element 9. C# currently does not allow you to change this behavior.

Working with Arrays

Arrays are a powerful feature in every programming language but even more useful in C#. This is because arrays are a type of object in C#. Like other objects, arrays have methods and properties. When you understand some of these characteristics of the array class, you can do many interesting things with arrays.

Using the Array Demo Program to Explore Arrays

The Array Demo program featured in Figure 8.5 puts the array class through its paces, showing off some interesting capabilities. The program features a list box control for displaying the array (lstOutput), a pair of option buttons (optSorted and optUnsorted) for determining whether the array is sorted, and a button (btnForEach) for demonstrating the foreach loop. Notice that the search button (btnSearch) and its associated text box (txtSearch) are disabled. The user can see them, but they are not currently functional. This is because the searching operation I'll be demonstrating works only with sorted arrays. When the array is sorted, I'll make these controls available to the user.

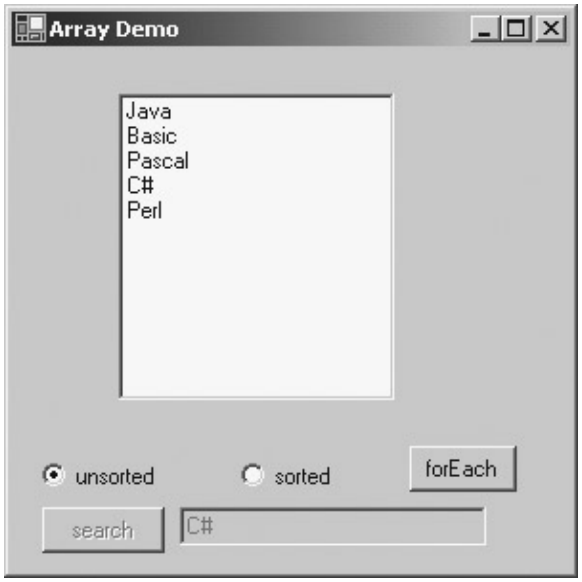


Figure 8.5: An array of programming language names is displayed in a list box. The array is not in any particular order.

Building the Languages Array

The Array Demo program is (not surprisingly) closely related to an array. For lack of anything more interesting, I chose to create an array of programming language names. Because I use the array throughout the program, it is declared at the class level:

```
string[] languages = {
  "Java",
  "Basic",
  "Pascal",
  "C#",
  "Perl",
};

string[] sortedLangs = new string[5];
showUnsorted();
```

If you examine the code, you will see that I created the array a little differently than in the Counter program. If you know exactly what elements are going to go in the array when you create it, you can "preload" the array, as I did in this example. Notice the use of braces to indicate the beginning and end of the array definition. Also, because arrays can be large, I chose to put each element on a different line of the code listing.

The sortedLangs variable is another string array of five elements. This array will hold the sorted version of the languages array.

The last line calls a method that will show the unsorted version of the array in the list box. The details of that method are explained in the next section.

Sorting the Array

When you have an array of data, you usually need to sort it. Sorting an array by hand can be difficult. Computer science instructors love to drone on endlessly about the subject. In C#, however, you usually don't have to worry about the details of sorting because the array class already has an efficient sort method built in. Figure 8.6 shows the array after it has been sorted.

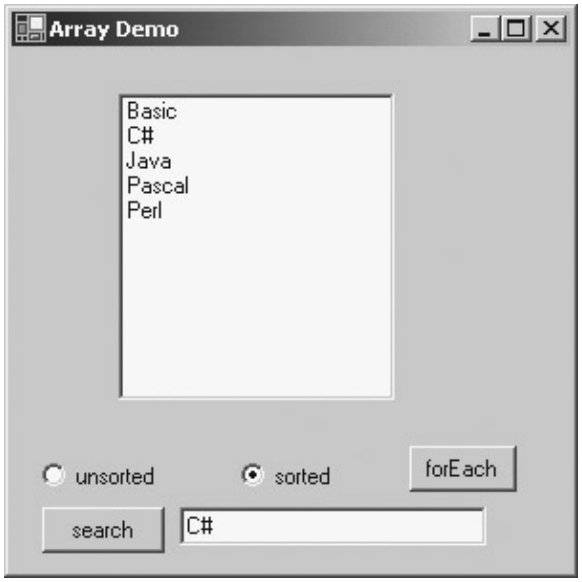


Figure 8.6: When the user chooses the sort option, the array appears in alphabetical order.

Trick The array is invisible to the user. Arrays are data structures stored in the computer's memory. For the sake of demonstration, I've copied the array over to the list box so that you can see what's going on "under the hood."

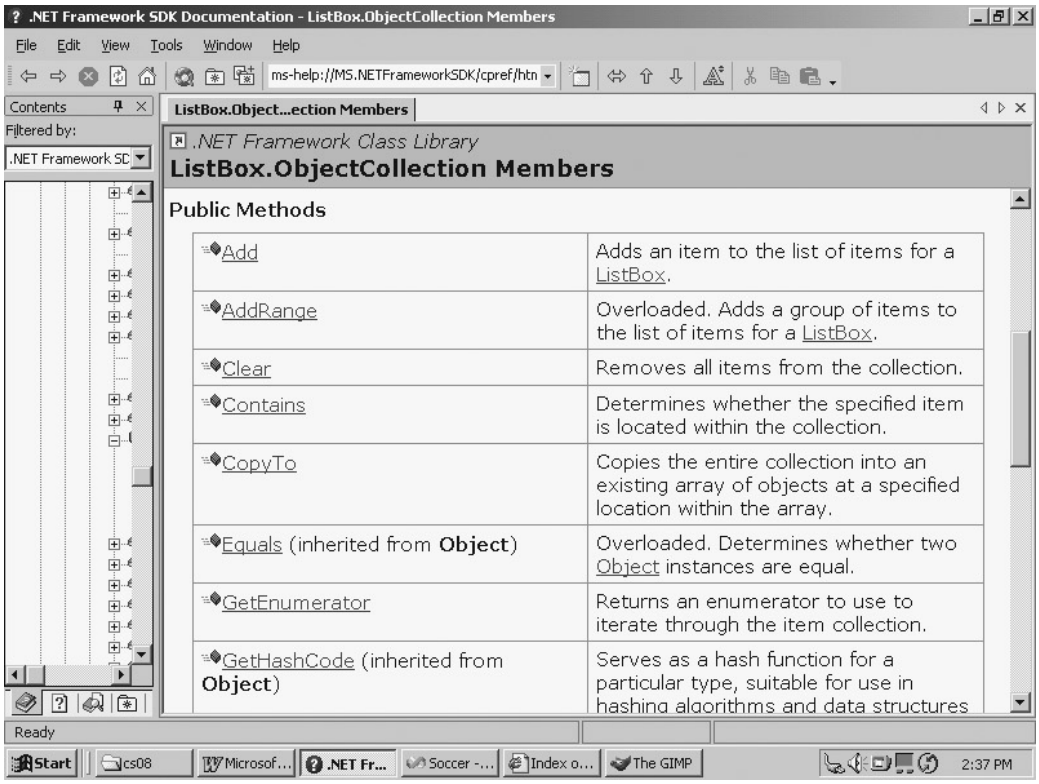
The user can choose the sorted or unsorted view of the array via the option buttons. Option buttons, as you recall, are handy when only one out of a group of choices should be selected at a time. I added code to the default event of the option buttons to manage the array sorting.

Creating the ShowUnsorted() Method

Because the unsorted array needs to be displayed at the beginning of the program and whenever the unsorted option is selected, I decided to make it a method:

```
private void showUnsorted(){
  //displays the array in the list box in its default state
  lstOutput.Items.Clear();
  lstOutput.Items.AddRange(languages);
} //end showUnsorted
```

The ShowUnsorted() method begins by clearing the list box. You might expect the list box class to have a Clear() method (I expected that, anyway). It doesn't, exactly. Instead, the Clear() method belongs to the Items property, which is an instance of the Listbox.ObjectCollection class. Although this seems complex, it's actually a good thing because the ObjectCollection class is a handy class with neat features for adding elements to a list box, clearing the elements from the list box, and more. Figure 8.7 shows the help screen for this class.


.NET Framework SDK Documentation - ListBox.ObjectCollection Members
File Edit View Tools Window Help
ms-help://MS.NETFrameworkSDK/cpref/htn
Contents
Filtered by:
.NET Framework SD
ListBox.Object...ection Members
.NET Framework Class Library
ListBox.ObjectCollection Members
Public Methods
Add
Adds an item to the list of items for a ListBox.
AddRange
Overloaded. Adds a group of items to the list of items for a ListBox.
Clear
Removes all items from the collection.
Contains
Determines whether the specified item is located within the collection.
CopyTo
Copies the entire collection into an existing array of objects at a specified location within the array.
Equals (inherited from Object)
Overloaded. Determines whether two Object instances are equal.
GetEnumerator
Returns an enumerator to use to iterate through the item collection.
GetHashCode (inherited from Object)
Serves as a hash function for a particular type, suitable for use in hashing algorithms and data structures
Ready
Start
cs08
Microsof...
.NET Fr...
Soccer -...
Index o...
The GIMP
2:37 PM
Figure 8.7: The methods of the Array object make it easy to work with, especially if you are working with some kind of array.

Specifically, the Clear() method is used to clear all elements from the list box, and the AddRange() method is used to copy an entire array to the list box. If you want to add only one element at a time, use the Add() method instead. As you can see, the list box control is like a visual wrapper around an array. It's easy to copy array values to the list box and get an array from the list.

Handling the Unsorted Option

When the user clicks the unsorted option, the optUnsorted_CheckChanged() method is automatically called. The code for this method is very straightforward:

```
private void optUnsorted_CheckedChanged(object sender,
System.EventArgs e) {
  btnSearch.Enabled = false;
  txtSearch.Enabled = false;
  showUnsorted();
} // end optUSorted
```

If the array is unsorted, the search method will not work, so I disabled the searching controls. Doing this is a good idea because it prevents the user from getting an error. When the searching controls are turned off, the method calls the showUnsorted() method, which copies the original (unsorted) languages array to the list box.

Handling the Sort Option

If the user chooses to sort the array, an event handler of the optSorted object does the job:

```
private void optSorted_CheckedChanged(object sender,
System.EventArgs e) {
  languages.CopyTo(sortedLangs,0);
  Array.Sort(sortedLangs);
```

```
  lstOutput.Items.Clear();
  lstOutput.Items.AddRange(sortedLangs);
  btnSearch.Enabled = true;
  txtSearch.Enabled = true;
} // end optSorted
```

Because the user might want the original array again, I decided to sort a copy of the original. The languages variable is an array, and the array class has a CopyTo() method. This method takes two parameters. First, it needs the name of an array to copy to. Also, it needs to know which element to start with. The target array should have the same type as the original and should have at least as many elements as the source array. In this case, I called the sorted array sortedLangs. The Array.Sort() method provides an easy way to sort the array alphabetically. After the array was sorted, I cleared out the list box and copied the sortedLangs array to it.

The user should be allowed to search the array now because it is sorted, so I enabled the button and text box for searching.

Trick If it bugs you to have elements on the screen that are visible but not available to the user, you can make them completely visible and invisible by manipulating the Visible property instead of the Enabled property. Some argue that leaving controls visible but disabled gives the user a clue that searching the array is possible, but not in the program's current state.

Searching for an Element

If an array is large, searching through it to find where a particular element is in the array can be very helpful. Although the arrays used in this demo are not large enough to need this functionality, it is still useful, especially because the Array object has a very nice search method built in, as illustrated in Figure 8.8.

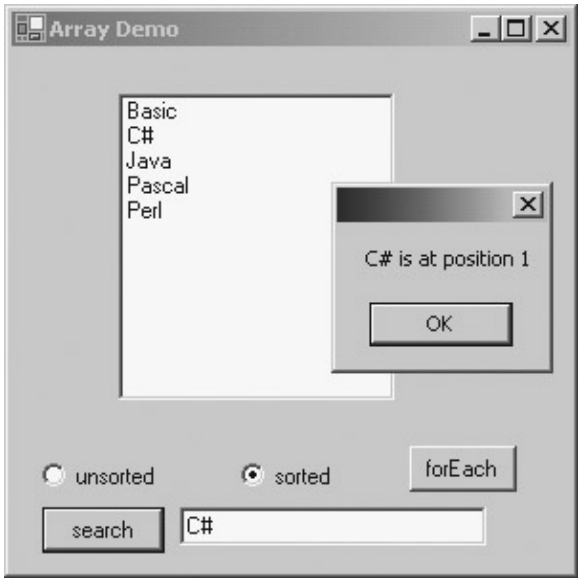


Figure 8.8: The user can search for an element to see where it is in the array.

Searching efficiently for elements in an array can be surprisingly difficult, especially if the array contains hundreds or thousands of elements. One of the most powerful types of search algorithms is the *binary* search. It works only on sorted arrays but can quickly find where a value is in even very large arrays. This binary algorithm is the one implemented in the BinarySearch() method. Searching for a value in the Array Demo program might seem trivial because the underlying array has only five elements. Searching and sorting arrays are much more critical operations when the array is larger, but these operations work identically on smaller arrays, too.

The search operation happens when the user clicks the Search button:

```
private void btnSearch_Click(object sender,
  System.EventArgs e) {
  int theIndex;
  string message;
  theIndex = Array.BinarySearch(sortedLangs,
    0,sortedLangs.Length, txtSearch.Text);
  if (theIndex < 0){
    message = "not found.";
  } else {
    message = sortedLangs[theIndex];
    message += " is at position ";
    message += theIndex.ToString();
  } // end if
  MessageBox.Show(message);
} // end btnSearch
```

The method has two variables. theIndex stores the results of the search, and message holds a message to the user regarding the success or failure of the search.

I used the BinarySearch method of the Array class to search for the contents of the txtSearch text box in the sortedLangs array. The BinarySearch method also requires a starting number and length. I specified that the search should start at element 0 and continue for the length of the array. If the search value is not found, the algorithm returns a negative value, so I tested for the not found condition. If the result of the search is negative, I inform the user that the search was not successful. If the result is positive, the program returns the location of the value in the array.

> **Trap** Note that the elements are counted beginning with 0. Even though Basic is the first item in the sorted array (at least, to human reasoning), it is reported as element number 0.

Using Foreach with Arrays

You learned how to use the foreach loop in Chapter 3, "Loops and Strings: The Pig Latin Program." However, my explanation of the loop in that chapter is weak because the foreach loop is closely related to arrays, which you hadn't learned yet. Figure 8.9 illustrates part of what happens when the user clicks the forEach button.

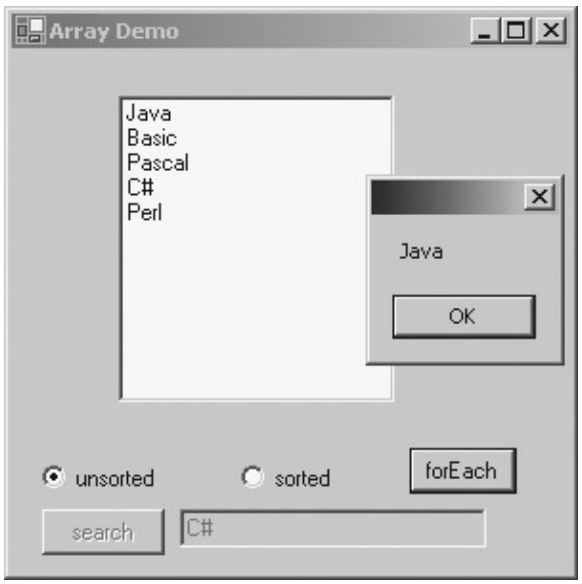


Figure 8.9: When the user clicks the forEach button, the elements of the array are shown one at a time.

Wanting to step through the elements of an array one at a time is common. Arrays and for loops are a natural combination because doing something to each element of an array one at a time is so common. Although you can use a for loop for this purpose, the foreach loop makes examining an array, element by element, even easier to do.

```
private void btnForEach_Click(object sender,
  System.EventArgs e) {
  //demonstrates the foreach loop used with arrays
  foreach (string theLang in languages){
    MessageBox.Show(theLang);
  } // end foreach
} // end btnFor
```

The foreach loop is a special variant of the for loop. It requires a local variable of the same type as the array and the array name. In this case, theLang is a string that holds each value in the array. Because it's a string array, theLang must be a string value. The variable name must be followed by the keyword in, followed by the name of the array. The loop occurs once for each element of the array. Each time through the loop, theLang has a different value picked from the array.

Trick The reason this works in the Pig Latin game is this: When I used the Split() method on a string variable, it converted that variable to an array of strings. I could then use the foreach loop to step easily through that array. Closure at last…

Creating Tables with Two−Dimensional Arrays

The type of array you've seen so far in this chapter is used whenever you need to work with a list of data. Sometimes, however, you are more interested in more complex information. When you solve complex problems without a computer, you often find yourself keeping information in two−dimensional tables. C# supports a special two−dimensional array that allows you to work with this kind of information.

Designing the Soccer Game

The Soccer program you saw at the beginning of this chapter relies on a table. If you analyze the Soccer program, you will see that all the action revolves around the user's clicking the various players. Somehow the computer has to know the likelihood that a pass from one player to another will succeed. Figure 8.10 shows a program that illustrates the likelihood of any one player's passing to another.

Figure 8.10: The halfback should complete a pass to the wing 80 percent of the time.

Hint Usually I use the terms *player* and *user* interchangeably, but in this chapter, I have decided to use these terms more carefully. The soccer game has entities called *players,* and I have chosen to use that term only when I'm referring to these elements inside the game. I use the term *user* to describe the person playing the game.

Because this is the most critical part of the Soccer game that concludes this chapter, you should learn the basic concepts in a simpler program first, such as the Shot Demo illustrated in Figure 8.10.

As you can see in the Shot Demo program, the user can select one player to kick the ball and one player (or shot) as the target. Whenever the user makes a selection in one of the list boxes, the percentage likelihood this shot will succeed is displayed. The user can then press the kick button, which will succeed at the indicated percentage, to get a feel for how often the indicated percentage succeeds.

Solving a Subset of the Problem

One way to set this up would be to think about a certain player at first. For example, take the fullback, who generally stays near his own goal in a defensive position. It is very likely that the fullback will complete a pass to the goalie or halfback, because these two players are usually in close proximity. It is less likely that the fullback will complete a pass directly to the wing or center and highly unlikely that an attempt on the opponent's goal will succeed from the fullback's position. You could encode this information in a simple chart like Table 8.1.

Table 8.1: The Likelihood of Success from the Fullback Spot

Goalie	Halfback	Wing	Center	Shot
80%	80%	60%	40%	2%

Table 8.1 summarizes the chances of success for a shot or pass from the fullback to other members of the team (or the opposing goal). Although there is absolutely nothing scientific about these values, they are an approximation of the options facing a fullback during a soccer game. Having percentages is nice from a programming point of view because generating a random value that will approximate any percentage is very easy. For example, if you want to set up a condition that will be

true 60 percent of the time (to simulate a pass from the fullback to the wing), you could use something like this:

```
Random roller = new Random();
if (roller.nextDouble() < .6){
  MessageBox("Good!");
} else {
  MessageBox("No good.");
} // end if
```

The nextDouble() method produces a random number between 0 and 1. That value will be less than 0.6 about 60 percent of the time. If you have percentages for the likelihood of various situations, writing conditional statements to test whether those situations occur is easy.

Adding Percentages for the Other Players

It would be possible to encapsulate all the information about the fullback's options in a normal array, but sometimes other players have the ball, too. You can build a more complex table that tries to show all the possible choices in the game. Table 8.2 illustrates one such chart.

Table 8.2: Percentages For All Plays

Kicker	Goalie	Fullback	Halfback	Wing	Center	Goal
Goalie	–1	.8	.6	.4	.1	.01
Fullback	.8	–1	.8	.6	.4	.02
Halfback	.8	.8	–1	.8	.6	.03
Wing	.8	.8	.8	–1	.8	.04
Center	.8	.8	.8	.8	–1	.2

Table 8.2 extends Table 8.1 to consider the likelihood of every opportunity offered to the player during the Soccer game. In this table, the percentages are written as double values (so *80%* is written as *.8*, for example). Also, I used a *–1* to indicate an impossible shot. I wanted to require the user to keep moving the ball around the field so that if he attempts to pass from a player *to* that same player, it will always fail (and in the final game, the opposing team will get a chance to score).

There is absolutely nothing scientific about the percentages I chose. I pulled them completely out of the air as a starting point. When the mechanics of the game are working, it will probably be necessary to tweak the values in this table.

Setting Up the Shot Demo Program

The Shot Demo program uses a special form of array to duplicate the information from Table 8.2 in the computer's memory. The class–level code includes a new type of array to hold the data:

```
double chance = 0d;

private double[,] shotChance = {
  {-1, .8, .6, .4, .1, .01},
  {.8, -1, .8, .6, .4, .02},
  {.8, .8, -1, .8, .6, .03},
  {.8, .8, .8, -1, .8, .04},
  {.8, .8, .8, .8, -1, .2},
};
```

Because the program uses percentages extensively, the double type is used to handle all percentage values. The player's likelihood of succeeding at a shot is stored in the chance variable. The shotChance is an array of doubles. However, you can see that its structure is fundamentally different from the other arrays you've seen so far. The data I'm trying to encapsulate in shotDemo comes from a two–dimensional table, so I store it in a two–dimensional array. You set up an array to have two (or more) dimensions by adding a comma (or more) to the brackets that follow the array type. You can predefine the values of the array if you know what the array will contain when you are initializing it. Each row is enclosed in braces, and the entire structure is enclosed in another set of braces. The rows, just like the data inside the rows, are separated by commas.

You can also set up the array without initializing it. If you wanted to make a 3–by–4 double array, for example, you could use a line such as

```
private double[,] myArray = new double[3,4];
```

When you have a two–dimensional array, you refer to it using two indices. For example, shotChance[3,2] refers to the third column, second row of the shotChance array, which is the value .8. (Don't forget that the computer starts counting with 0, so the *third* column is what people would consider the fourth column.)

In the Real World

Beginning programmers almost never use an array in this situation. Instead, they shy away from data structures and look at a solution that uses control structures. This is usually a mistake. It would be possible to use a series of if–elseif–endif statements or a nested switch structure to get the same behavior as the array technique I'll show you here, but that technique would require between 75 and 100 lines of code. A lot of repeated code means many places where things can go wrong and a major headache if you find that you need to tweak your logic. By taking the time to think carefully about a data structure, the 75 lines of code can be shrunk to 4. It's an amazing improvement in efficiency, and those four lines of code are easier to fix and modify. Experienced programmers often look for control structures that are more complex in the short run but pay off in code complexity for the long run.

Hint There's no reason to stop at two dimensions. It's possible to build arrays with four, five, or as many dimensions as you like. Usually, programmers don't need to make arrays much larger than three dimensions because they can store objects in the arrays, which allow for even more complex and flexible data.

Setting Up the List Boxes

The user will use the list boxes to control the program, so setting them up properly is very important. In particular, the elements in the lists must go in the correct order. The lstKicker list box represents all the various kickers. Each player can be a kicker, but not the opponent's goal (it is strictly a target), so the list box is filled with the names of the positions. I took special care to associate the names with the rows of the original table (Goalie, Fullback, Halfback, Wing, and Center). The lstTarget list box contains all the various targets. Each player position is a potential target, as is the opposing goal, so this list box has six elements, corresponding to the columns in the original table.

To ensure that the list boxes would begin with legal values, I added code to the constructor to initialize the list boxes:

```
lstKicker.SelectedIndex = 0;
```

```
lstTarget.SelectedIndex = 1;
```

The SelectedIndex property is used to determine which element in a list box is currently selected. In this case, I chose to preset lstKicker at the 0 element (goalie) and lstTarget at element 1 (fullback).

Using a Custom Event Handler

After I designed the visual interface of the Shot Demo program, I started to think about the event handling. I soon realized that the exact same code should happen whenever either list box is changed, because the program will always need the values of both list boxes to determine how likely the shot will be. There are a couple ways to handle this. The most straightforward is to write code in the event handler for one list box and then copy and paste the code to the other. However, if you have to change the code, you need to change it in two places. Another solution would be to build a special method that evaluates the list boxes and to call that method from both event handlers. C# has another nice trick, though. Rather than have C# make event handlers automatically, you can build your own method and designate it to be an event handler. All event handlers must have two parameters. To be consistent with the automatically generated code, I used the same names as the default names for the sender and EventArgs variables.

The declaration for the new changeStatus method looks like this:

```
public void changeStatus(object sender,
  System.EventArgs e){
```

After you write a method that you want to use as an event handler, you connect it to the events it should respond to. This is easily done with the Visual Designer in the events panel of the Properties window. Figure 8.11 shows assigning a custom method to a list box's events.

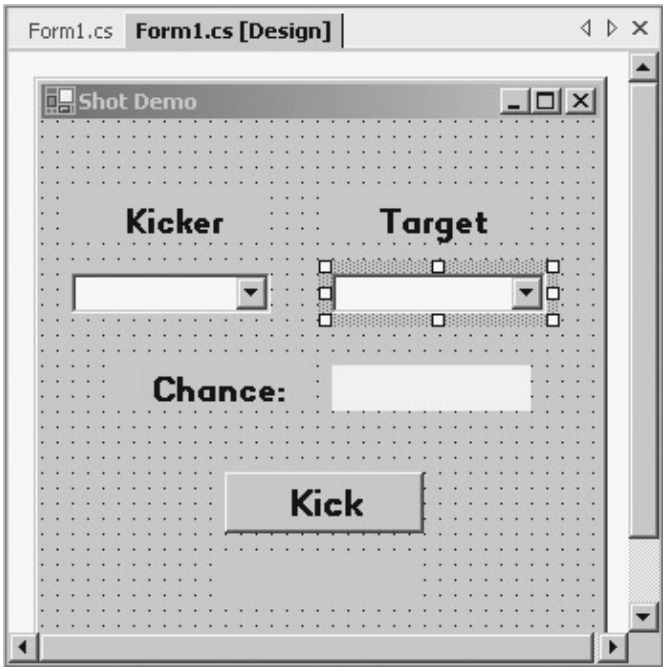


Figure 8.11: The drop–down menu is automatically populated with all methods detected by the IDE. The Visual Designer can detect event handlers by their parameters, so it automatically puts them in the list box associated with an event. In this way, you can assign several events to the same method, if you like.

Writing the changeStatus() Method

The behavior of the changeStatus() method is not difficult. It starts with two int variables representing the kicker and the target. The kicker variable contains the selectedIndex property of the lstKicker list box. Likewise, the target variable is extracted from the lstTarget list box. The selectedIndex property returns an integer showing which element in a list box is currently selected.

```
public void changeStatus(object sender,
  System.EventArgs e){
  //handles a change in either listbox
  int kicker = 0;
  int target = 0;

  //get the kicker
  kicker = lstKicker.SelectedIndex;

  //get the target
  target = lstTarget.SelectedIndex + 1;

  chance = shotChance[kicker, target];
  lblChance.Text = chance.ToString();

}  // end changeStatus
```

When there is a legitimate value in the kicker and target variables, the method looks up a value in the shotChance array and assigns this value to the chance double. It also sends a text version of this value out to the text box to tell the user which value was extracted.

Kicking the Ball

The last part of the Shot Demo program involves calculating whether a shot is successful. If the Shot Demo program can demonstrate this behavior in this simpler environment, transferring this code to the more complex Soccer program should be easy:

```
private void btnKick_Click(object sender,
  System.EventArgs e) {
  Random roller = new Random();
  double toHit = roller.NextDouble();
  if (toHit < chance){
    lblResult.Text = "Hit";
  } else {
    lblResult.Text = "Miss";
  } // end if
} // end btnKick
```

In the Real World

Two−dimensional arrays are often called *lookup tables* because they can be used just as I have done in this program. I stored information in a table and then looked it up by the row and column. You can use two−dimensional arrays any time what you are working on requires you to look up an item in a table. In nongame programming, lookup tables are frequently used for tasks such as determining shipping rates, sales tax, or any other value that might commonly be stored in a table.

Whenever the user clicks the button, this method springs into action. Like any other method that

uses random numbers, I created an instance of the Random class. The toHit variable comes from the Random object's nextDouble() method. Remember, the program will be comparing against a double value from the lookup table, so it needs to be a double as well. If toHit is less than chance (which was set when the user made a selection from one of the list boxes), the method reports a hit. Otherwise, the result label gets the value *Miss*.

Designing Programs by Hand

Arrays are powerful, as you see in the examples in this chapter. You can even make arrays out of complicated types such as classes and even screen components such as buttons and labels. Arrays of controls are especially useful, but sadly, the Visual Designer does not provide an easy way to work with them. The Soccer program will use such an array to hold the player images as they move around the field. Although the Visual Designer is very convenient, it can't always create the exact type of program you need, so you have to know how to build a form without the designer.

Examining the Form by Hand Program

The Form by Hand program shown in Figure 8.12 looks much like any other Windows program. The interesting thing about this form is that it was built without the Visual Designer. Figure 8.13 shows the Visual Designer for the completed program.

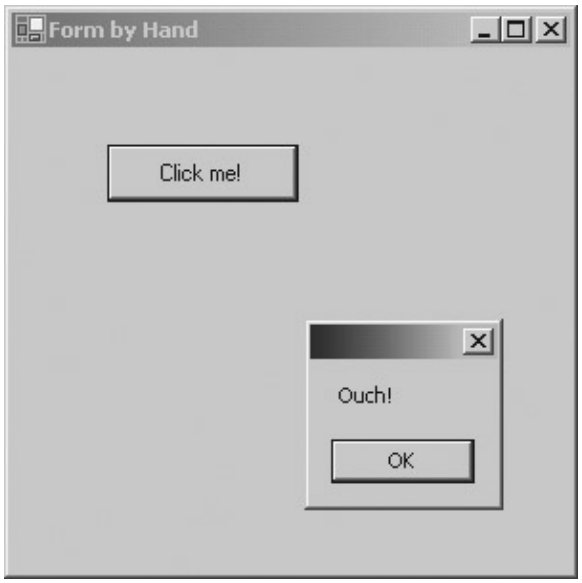


Figure 8.12: The form has a button and responds to the button's click event.

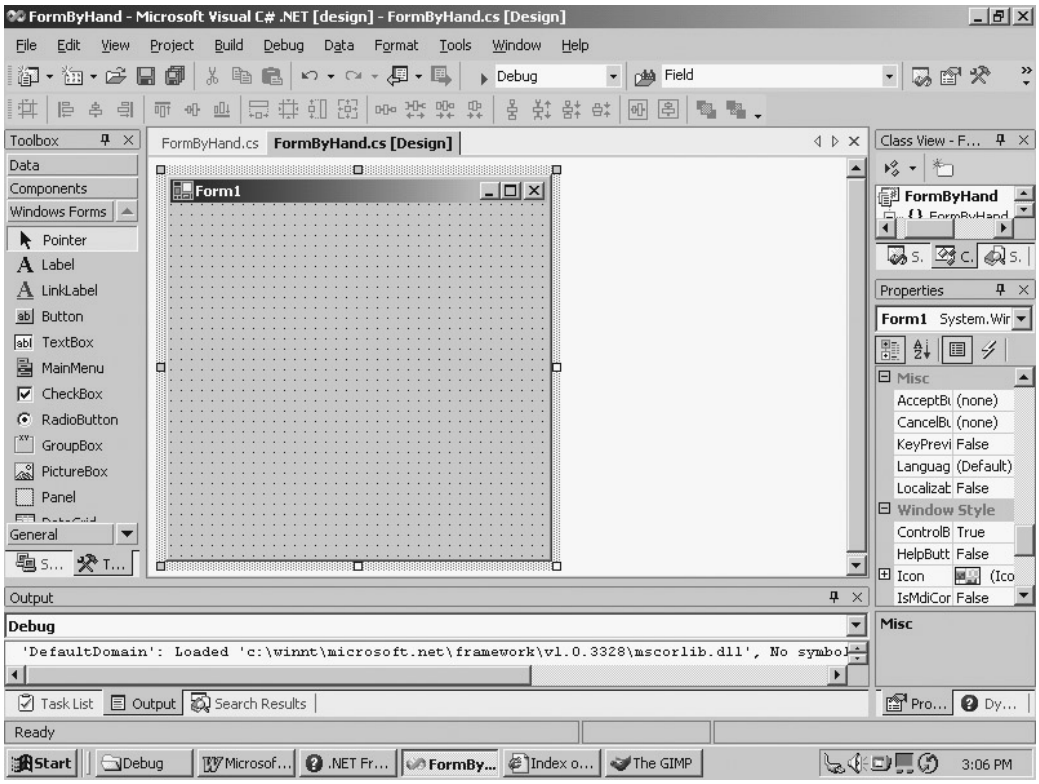


Figure 8.13: The Visual Designer shows no components at all!

It is possible (and sometimes desirable) to add controls to a program without using the Form Designer. Even when you do use the designer, you should know what it's doing behind the scenes. I added the button and set up its event behavior entirely with code, without using the designer at all.

Adding Components in the Constructor

The constructor is a good place to add various components because it happens early. The Visual Designer places all its code in the InitializeComponent() method. To make sure that your code takes precedence over any automatically generated code, place your code in the constructor after the call to InitializeComponent():

```
public FrmByHand()
{
  InitializeComponent();

  Form1.Title = "Form by Hand";
  Button myButton = new Button();
  myButton.Location = new Point(50, 50);
  myButton.Size = new Size(100, 30);
  myButton.Text = "Click me!";
  this.Controls.Add(myButton);

  myButton.Click += new EventHandler(myButton_Click);
}  // end constructor
```

The form itself is simply an object of type Form, so you can change its properties, just like any other object. The button is, likewise, simply a class. You can assign a point as the size property of the button (or any other component), and you set the size property by assigning a Size object. To determine the appropriate size and location, you can use graph paper, as well as experiment by trial and error. The program then sets the text to the button so that it has an appropriate caption. To add the control to the form, you add it to the form's controls collection. This is much like adding an item

to a list box, but the controls collection is designed to accept controls. The final task for the constructor is to register an action listener to the button. The last line of the method accomplishes this task. myButton.Click refers to the click event of the myButton object. You can use the += operator to add an event handler to this event.

Hint You can add multiple handlers to the event, if you like, so that clicking the button calls several methods. Doing so can cause problems, though, because it is unclear which order the methods would evaluate.

An event handler is a class. To make one, you call its constructor with the name of the method you will use to respond to this event. That event method (like all event methods) will need to have an object and an EventArgs parameter.

Responding to the Button Event

Because I used myButton_Click as a parameter when I built my EventHandler, I have to create a method named myButton_Click:

```
public void myButton_Click(object sender,
System.EventArgs e){
  MessageBox.Show("Ouch!");
} // end myButton
```

The only way this method differs from any other event handler is that the Visual Designer did not create it. I had to write the code by hand, including the parameters. Other than that, the method is no different from many you have written by now.

Building the Soccer Program

You now have all the skills you need to build the soccer game introduced at the beginning of the chapter. The game might appear complex if you look at the entire thing at once, but when you break it into its constituent parts, the soccer game isn't nearly so intimidating.

Setting Up the Variables

As usual, the variables tell you a lot about the program. The Soccer program features a large number of variables declared at the form level, but they are not tricky.

Variables about the Players

The first group of variables is used to give names to the various positions:

```
//set up constants for players
private int GOALIE = 0;
private int FULLBACK = 1;
private int HALFBACK = 2;
private int WING = 3;
private int CENTER = 4;
private int SHOT = 5;

//array for player names
private string[] playerName = {
  "Goalie",
```

```
  "Fullback",
  "Halfback",
  "Wing",
  "Center",
  "Shot"
};
```

Many times throughout the game code, it is necessary to determine which player you're talking about. Rather than memorize that the goalie is player 0 and the shot (opposing goal) is player 5, I created special variables to hold these names. The use of all uppercase is traditional when you create a variable like this, which you don't intend to change throughout the program. You will see how useful these variables are later on as you read through the code.

Likewise, having a string array of strings associated with an array is sometimes handy so that you can easily tell the user what's going on. The playerName array will typically be used when I want to tell the user something about a player.

Game Management Variables

The next batch of variables is used to control the general flow of the game. These variables are the lifeblood of the game:

```
//primary game variables
private int playerScore = 0;
private int oppScore = 0;
private int timeLeft = 600;
private int currentPlayer = GOALIE;
private int nextPlayer = FULLBACK;

//data structure for likelihood of shot
private double[,] shotChance =
{{-1, .8, .6, .4, .1, .01},
 {.8, -1, .8, .6, .4, .02},
 {.8, .8, -1, .8, .6, .03},
 {.8, .8, .8, -1, .8, .04},
 {.8, .8, .8, .8, -1, .2},
 {.8, .8, .8, .8, .8, -1}};
```

The first batch of variables in this group holds typical game variables, such as the player and opponent score, time left in the game (in 10ths of a second), which player currently has the ball, and which player is set to receive the ball. Notice the use of GOALIE and FULLBACK. Because I set these variables earlier, I could use them here to make my intentions much clearer than if I'd just put the numbers 0 and 1.

The Picture Box Arrays

The players zipping around on the field are actually picture boxes. These picture boxes are the trickiest part of the game because there are so many of them. To manage them by hand would be crazy, so I built two arrays. You can make an array out of any sort of variable, including picture boxes. Of course, the Visual Designer does not support placing arrays of controls. I'll have to place them by hand, but it's not difficult, as you will see:

```
//custom picture box arrays for players
private PictureBox[] picPlayer = new PictureBox[6];
private PictureBox[] picOpp = new PictureBox[5];
```

Notice that I made two arrays. The picPlayer array deals with the players that the user can click (all the players on the yellow team and the red goalie). The picOpp array contains picture boxes that handle all the players on the red team except the red goalie.

Note that the distinction between the player and opponent arrays is *not* which team the picture box represents. The red goalie is part of the picPlayer array because it shares a lot of functionality with the players on the yellow team. It can be clicked, and it can be a target to shoot at. None of the other red teams need to respond to click events, and they don't do anything but move around randomly. The picture boxes that need to respond to events are stored in one array (picPlayer), and those that do not need to respond to events are stored in the other array (picOpp).

Building visual controls with the Form Designer is much easier, so I built everything but the picture box arrays, using the designer:

```
//controls built by designer
private System.Windows.Forms.Label lblAnnounce;
private System.Windows.Forms.Timer timer1;
private System.Windows.Forms.Label lblTime;
private System.Windows.Forms.Label lblPlScore;
private System.Windows.Forms.Label lblOppScore;
private System.Windows.Forms.ImageList myPics;
private System.Windows.Forms.Panel pnlField;
private System.ComponentModel.IContainer components;
```

The lblAnnounce label delivers a play–by–play account of the game. It announces each pass and shot, helping the user figure out what's going on in the game.

The game features a standard timer named lblTime. The timer's interval is set to 100 milliseconds, which means that the game runs at 10 frames per second.

The lblPlScore and lblOppScore labels are used to display the player and opponent scores, respectively.

The program features an ImageList control named myPics, which contains seven images (even though I didn't use them all in the final game). The images include a yellow player without the ball, a yellow player with the ball, a red player without the ball, a red player with the ball, a yellow goalie and red goalie, and a large red square I used when debugging. Figure 8.14 shows the various images associated with the image list control.

Index	Image
0	
1	
2	
3	
4	
5	
6	

Figure 8.14: The images stored in the image list control.

Hint That large red square turned out to be handy, even though I didn't include it in the final game. At one point, I had a player just sitting there, not moving. I couldn't figure out which player it was, so I was unsure how to fix it. One by one, I set the image of each player to the red box and ran the program. After the offending player was set as the red box, I knew which player was having trouble, and I found the problem in the code. Sometimes you have to be inventive in your debugging strategies.

The field itself is a panel, with an image of a soccer field attached. I added all the player picture boxes directly to the panel rather than to the form itself because the panel's dimensions are a handy way to determine the field's borders. All the other controls (the labels for timing and score) are added to the form itself.

Examining the Constructor

The Soccer program is heavily dependent on custom controls. You might expect a lot going on in the constructor, and you'd be right. However, when you look at the constructor code, you will find it very sparse:

```
public frmSoccer()
{
  //
  // Required for Windows Form Designer support
  //
  InitializeComponent();

  setupPlayers();
  setupOpp();
  setupGoalies();

} // end constructor
```

There is so much work for the constructor to do that I decided to break it into smaller sections and send off some of the work to methods. (Encapsulation! I *love* encapsulation!) This makes it very clear what the big picture assignments are in the constructor and enables you to separate the strategic problem (what kinds of things have to happen in the constructor) from the tactical problems (such as how you set up each picture box array). As you can see, the constructor has four major jobs.

The first is to call InitializeComponent(), which manages all the components created in the Visual Designer. Even though I built some controls by hand, calling InitializeComponent() is still important. If you're going to add other code to the constructor, that code should come after the InitializeComponent() call.

All the other jobs in the constructor involve setting up the picture box arrays. The array of clickable picture boxes has to be set up. The setupPlayers() method will handle this chore. The opponents have to be set up as well. Their setup is a little different, so it will be handled in a different method, named setupOpp(). Finally, the two goalies have slightly different behavior than the other picture boxes, so they need their own setup routine.

Setting Up the Players

The picPlayer array of picture boxes is the user's main interface with this program. All the user's interactions with the program will come from clicking one of these players. Because the Visual Designer isn't good at arrays, you yourself have to set them up. It's a little tedious, but not difficult.

```
private void setupPlayers(){
      //setting up my own picture boxes
      int x;
      int y;
      //step through the players
      for (int i= 0; i < 6; i++){

        //standard control setup
        picPlayer[i] = new PictureBox();
        picPlayer[i].Size = new Size(25,25);

        //start all players in the center of the field
        //on the x (horizontal) axis
        x = (int)((pnlField.Width -
picPlayer[i].Width)/2);

        if (i > GOALIE){
          //move all players but goalie to a Y value
          //related to their position (center close to opp),
          //fullback close to own goal
          y = (int)(pnlField.Height/i);
          picPlayer[i].Location = new Point(x, y);
        } // end if

        //set up the image to the yellow no ball player
        picPlayer[i].Image = myPics.Images[2];
        picPlayer[i].SizeMode =
PictureBoxSizeMode.StretchImage;

        //use the tag field to store the player number
        picPlayer[i].Tag = i.ToString();

        //register the event listener
        picPlayer[i].Click += new
EventHandler(clickPlayer);

        //add the player to the panel
        pnlField.Controls.Add(picPlayer[i]);
      } // end for loop
    } // end setupPlayers
```

A lot is going on in this method, but the comments clearly describe the action. As I've noted before, for loops and arrays are a natural combination. In this case, they save quite a bit of work because I can tell the program to step through each player and perform the same work on it, rather than repeat the code six times.

The first major job of the method is to ensure that each member of the picPlayer array is initialized. It's important to notice that even though I've initialized the array, I still need to initialize each of its members by assigning it a picture box.

I set each picture box to be 25 by 25 pixels. This is small enough to hide my poor drawing skills but large enough to be visible, and it causes the players to be scaled properly on the screen. The image of a yellow player without a ball is image number 2 in my image list, so I set each player to that image. I made sure that the picture boxes were set to stretchImage mode so that the images would size correctly.

Trick If you're not a brilliant artist, all is not lost. The images for many games are actually tiny. I designed my images as 50 x 50 images and then drew them at 400–percent of their original size. My large drawings looked incredibly crude, but when I shrank them down to 25 x 25

pixels, they looked good. It's smart to get and learn a good graphics package (such as The Gimp, included on the CD–ROM that accompanies this book) so that you can create your own images without the headache of dealing with intellectual property rights.

Getting a good starting position for the images was tricky. Although the position of the images doesn't matter at all in terms of game play (at least, the way the game is set up right now), it is important that the players at least appear to be dispersed around the field. I decided to place all the players at the center of the field horizontally. If you subtract the player's width from the field width and then divide by 2, you get the appropriate placement to center the player horizontally. Getting a good vertical position took more thought because the players should start out in different parts of the field. The center should be close to the opponent's goal, and the fullback should be near the user's goal. I found that if I divided the field height by the player number, I came up with a good vertical placement for each player except the goalie. I excluded the goalie from this calculation with an if statement because it will be placed more carefully in a later method. Also, excluding the goalie is necessary because its player number is 0 and the computer will choke if you ask it to divide by 0.

Later on, I'll need to know which player is which, so I used a very handy property named the Tag property. The Tag is basically a free–for–all property you can use to store any kind of information you want. I'll store the player number in the tag property for use later on in the click event.

All the picPlayer controls must respond to a click event, so I registered an event handler to a method named clickPlayer (which I'll build shortly).

Finally, I added each picture box to the field panel.

Setting Up the Opponents

The opponents are much like the player array, but because they won't respond to events, they are simpler to set up:

```
private void setupOpp(){
  //set up opponents
  int x;
  int y;

  for (int i= 1; i < 5; i++){
    picOpp[i] = new PictureBox();
    picOpp[i].Size = new Size(25,25);
    x = (int)((this.Width - picOpp[i].Width)/2);
    y = (int)(pnlField.Height/i  - 30);
    picOpp[i].Location = new Point(x, y);
    picOpp[i].Image = myPics.Images[0];
    picOpp[i].SizeMode =
PictureBoxSizeMode.StretchImage;
    pnlField.Controls.Add(picOpp[i]);
  } // end for loop
} // end setupOpp
```

The size and position of each picture box require a formula very much like the one in setupPlayer(). I set each image to the red player without a ball, which is image number 0 in the image list. I added each picture box to the controls of the panel, but adding an event listener was unnecessary.

Setting Up the Goalies

Both the goalies are part of the picPlayer array, and both have already been set up, for the most part. However, they deserve special treatment because they are positioned differently than the other players and have different images. Note that the user's goalie is named GOALIE, and the opposing goalie is named SHOT.

```
private void setupGoalies(){
  //place goalies more carefully
  int x;
  int y;

  x = (int)((pnlField.Width -
picPlayer[GOALIE].Width)/2);
  y = pnlField.Height - picPlayer[GOALIE].Height- 20;
  picPlayer[GOALIE].Location = new Point(x, y);
  picPlayer[SHOT].Location = new Point(x, 0);
  picPlayer[GOALIE].Image = myPics.Images[5];
  picPlayer[SHOT].Image = myPics.Images[4];
} // end setupGoalies
```

I used the now familiar horizontal centering routine to get the horizontal placement of the goalies. I moved the user goalie right above the bottom of the field and placed the opposing goalie near the top of the field. I then assigned appropriate images to the goalies from the image list.

Responding to Player Clicks

The game's action comes in two places. The passage of time will move the various player images on the field, but this has no real bearing on the game. The most important actions come when the user clicks one of the player images. Each of these images was registered with the clickPlayer() method as its event handler. This method takes action based on the algorithm described in the Shot Demo program earlier in this chapter:

```
private void clickPlayer(Object sender,
EventArgs e){
  //happens whenever you click a player
  Random roller = new Random();
  double toSucceed;
  double myRoll;

  //figure out which player was clicked
  PictureBox thePic = (PictureBox)sender;
  int playerNumber = Convert.ToInt32(thePic.Tag);
  nextPlayer = playerNumber;

  //figure out how likely success is
  toSucceed = shotChance[currentPlayer,
nextPlayer];

  //roll a random double
  myRoll = roller.NextDouble();

  //Announce what's going on
  string message = "From: " +
playerName[currentPlayer];
  message += " to: " +
playerName[nextPlayer] + " ";
  message += toSucceed.ToString();
  lblAnnounce.Text = message;
```

```
  //look for success
  if (myRoll < toSucceed){
    goodShot();
  } else {
    badShot();
  } // end 'pass succceeds' if

} // end clickPlayer
```

This method uses a number of variables to get started. It will need to know which player currently has the ball and to which player the user intends to pass. I'll need to do a little technical gymnastics to squeeze out this information, but it is accessible. The currentPlayer variable was declared at the class level, and its value is preset to whichever player currently has the ball. The trickier part is to determine nextPlayer, because the user doesn't directly enter this information. Instead, he clicks a picture box. The problem is to determine which picture box the user clicked.

Fortunately, C# provides a couple good tricks for figuring out exactly this type of information. First, recall the sender object that is automatically a parameter of all event handlers. This parameter contains an object that represents whatever object triggered the event. In this case, the picture box that the user clicked will be stored in the sender variable. I cast the sender object as a picture box and stored it in the local variable thePic. Back when I created each picture box, I stored its ID in the Tag property. I can extract that property now and convert it to an integer to determine which player was clicked. That value is stored in nextPlayer.

When I know what currentPlayer and nextPlayer are, I can use them to look up the likelihood of success in the shotChance lookup table. The mechanics of this activity are the same as in the Shot Demo program. The lookup table returns a double value representing the likelihood that the given shot will succeed.

I then send a message to the announcer label of the shot being attempted and the likelihood this shot will succeed. Finally, I evaluate the shot with a random number and call a method based on the results. If the shot succeeds, control flows to the goodShot() method. If the shot fails, the badShot() method takes control.

Handling Good Shots

If the shot is successful, the next step depends on what the user was aiming at. A successful shot on the goal means that the user has scored. If the user was aiming anywhere else, the ball should be passed to that player.

```
public void goodShot(){
  //if the shot succeeded

  //check to see if it's a shot on goal
  if (nextPlayer == SHOT){
    playerScore++;
    updateScore();
    MessageBox.Show("Goooooaaaaallllll!!!!");
    setPlayer(HALFBACK);
  } else {
    lblAnnounce.Text = playerName[nextPlayer] +
" now has ball";
    setPlayer(nextPlayer);
  } // end 'scored a goal' if
} // end goodShot
```

The if statement checks whether the user is shooting at the goal. If so, nextPlayer is set to the same value as SHOT. The user's score increases by 1, and the updateScore() method is employed to send a report to the scoreboard. I added a message box to enhance the mood. Finally, I gave the ball to the halfback, using the setPlayer() method.

If the user is not shooting at the goal, another player is now the current player. I copied the value of nextPlayer over to currentPlayer and set the current player visually with the setPlayer() method.

Handling Bad Shots

If the shot did not succeed, the opposing team has the ball, and I'll give them the chance to score. This is the easiest way to control the difficulty of the game. As the code stands now, the opponent will score about 5 percent of the time the player misses a pass or shot. To make the opposing team better, change the rate to a larger number. To make the opponent weaker, use a smaller number.

```
public void badShot(){
  Random roller = new Random();
  double toSucceed;
  int playerNumber;

  lblAnnounce.Text = "Opponent gets ball...";
  //check for opponent goal
  toSucceed = roller.NextDouble();
  //change the following value to alter
game difficulty
  if (toSucceed < .05){
    oppScore++;
    updateScore();
    MessageBox.Show("Opponent Scores!!");
    setPlayer(HALFBACK);
  } else {
    playerNumber = roller.Next(5);
    lblAnnounce.Text += "recovered by " +
playerName[playerNumber];
    setPlayer(playerNumber);
  } // end opponent scores if

} // end badShot
```

When the opponent gets the ball, I create a random double and compare it to .05. Five percent of the time, the random value will be smaller than .05, and the opponent will score. If this happens, I increase the opponent's score, update the scoreboard, and send a message box to the user. I then give the ball to the halfback to restart play.

If the opponent's shot did not succeed, I randomly determine which player begins with the ball and set that player as the current player.

Setting a New Current Player

Setting a player is a critical part of the game. It happens whenever the ball changes hands (okay, feet). This is a very frequent occurrence in this game. Setting a new player has two main effects. First, I change the variable currentPlayer to the new player's number, and then I change the visual representation so that the user can see that a new player has the ball:

```
public void setPlayer(int playerNumber){
  //given a player number, shows that player as
```

```
having the ball,
  //all others not having the ball

  //set up the current player variable
  currentPlayer = playerNumber;

  //show no player with ball
  for(int i = 0; i < 5; i++){
    picPlayer[i].BorderStyle = BorderStyle.None;
    picPlayer[i].Image = myPics.Images[2];
  } // end for loop

  //reset goalie image
  picPlayer[GOALIE].Image = myPics.Images[5];

  //show current player holding ball
  picPlayer[playerNumber].BorderStyle =
BorderStyle.FixedSingle;
  picPlayer[playerNumber].Image = myPics.Images[3];
}  // end setPlayer
```

Before I set up the visual representation of the player with the ball, I ensure that the preceding player is shown without the ball. The easiest way to do this is to set *all* the players without the ball. A player without the ball is represented by image 2 of the image list and has no border. The goalie image (if he doesn't have the ball) is image 5 of the image list. When all the images are clear, I set the border style of the current player to a fixed single–line border, and I set the image to image 3 of the image list, which shows the player with the ball.

Trick At first, I just used the ball to indicate which player had the ball, but the ball is so tiny that it is hard to spot. I added the border as an easy way to determine which player currently has the ball. Many sports games do something similar to ease gameplay.

Handling the Passage of Time

The program has a timer control used to pace the game. When the timer ticks, three things happen. The clock is updated, showing how many seconds are left of playing time. The player picture boxes are moved randomly, and the opponent picture boxes are also moved randomly. It shouldn't surprise you that each of these three tasks is relegated to a method, the updateTime(), movePlayers(), and moveOpp() methods.

```
private void timer1_Tick(object sender,
System.EventArgs e) {

  updateTime();
  movePlayers();
  moveOpp();

} // end timerTick
```

Again, encapsulation comes to the rescue. The timer_tick() method calls three other methods to do all the work, but it's easy to see from the tick method exactly which tasks occur whenever the timer ticks.

Updating the Clock

The first task is to update the clock. This seems quite simple, and it is, but merely reporting how much time is left is not enough. At some point, the user will be out of time, and the game will have to

end. The code to handle the end of the game is called inside the updateTime() method:

```
private void updateTime(){
  //calculate time left
  timeLeft--;
  if (timeLeft <=0){
    timer1.Enabled = false;
    DialogResult playAgain = MessageBox.Show
      ("Game Over. Play Again?", "Soccer",
MessageBoxButtons.YesNo);
    if (playAgain == DialogResult.Yes){
      //start over
      playerScore = 0;
      oppScore = 0;
      timeLeft = 600;
      currentPlayer = FULLBACK;
      updateScore();
      timer1.Enabled = true;
    } else {
      // end game
      Application.Exit();
    } // end playAgain if
  } else {
    double totalSeconds = timeLeft/10;
    int minutes = (int) (totalSeconds /60);
    int seconds = (int) (totalSeconds % 60);
    string timeString = minutes.ToString() + ":
" + seconds.ToString();
    lblTime.Text = timeString;
  } // end if

} // end updateTime
```

The first order of business is to decrement the timeLeft variable. This variable started out at 600, and it will be decremented by 1 each time the timer ticks. Because the timer is set at an interval of 10 frames per second, timeLeft will be 0 after 60 seconds. If timeLeft is less than or equal to 0, time has expired. In this situation, I display a dialog box asking whether the user wants to play again. If so, I reset all the key variables (score, time left, and current player variables) and restart the timer. If not, I end the application.

If the value of timeLeft is larger than 0, I do some quick math to determine how many minutes and seconds are left and then send this value to the user.

Trick Notice how I use the % operator to determine the number of seconds. This is called the *modulus* operator. It comes in handy in a number of situations, but many people don't know about it. The modulus operator returns the remainder of a long division problem. For example, if totalSeconds had the value 63, minutes would get the value 61 divided by 60, or 1 with a remainder of 3. If you divide two integers, the result is an integer, which completely ignores the remainder value. You can use the modulus operator to get the remainder of a division problem.

The minutes and seconds are stored as integers, so I reformatted them to send them to the time label.

Moving the Players

Part of this game's appeal is the way the little players run around on the field. To be honest, the movement of the players has absolutely no effect on the outcome of the game as it is currently set up. However, the action does add appeal, and you could change the code so that the distance between a player and the goal is the major determinate of the likelihood the player will score. (Hmmm, sounds like another good exercise.) All player motion is entirely random, as you can see from the code:

```
private void movePlayers(){
  //move players

 int motion;
 Random roller = new Random();

  for (int i = 1; i < 5; i++){
    motion = roller.Next(11) - 5;
    picPlayer[i].Left += motion;
    motion = roller.Next(11) -5;
    picPlayer[i].Top += motion;

    //check for boundaries
    if (picPlayer[i].Left < 0){
      picPlayer[i].Left = 0;
    } else if (picPlayer[i].Left +
picPlayer[i].Width > pnlField.Width){
      picPlayer[i].Left =
pnlField.Width - picPlayer[i].Width;
    } else if (picPlayer[i].Top < 0){
      picPlayer[i].Top = 0;
    } else if (picPlayer[i].Top +
picPlayer[i].Height > pnlField.Height){
      picPlayer[i].Top =
pnlField.Height - picPlayer[i].Height;
    } // end if

  } // end for loop
} // end movePlayers
```

I set up a for loop to go one at a time through each player (except the goalies—they don't move).

I used a random integer named motion to determine how much the player would move. I played with the motion variable to get exactly the effect I was looking for. The solution I settled on calls for grabbing a number between 0 and 10 and then subtracting 5 from that number. This results in a random value between –5 and 5.

> **Trap** My first thought was to use the two–integer version of the random object's Next() method to extract a value in the range I wanted. However, when I experimented with that approach, I found that the numbers were not as random as I wanted. They tended to be negative far more often than positive. This caused all my players to drift toward one corner of the field, which was not the behavior I was looking for. There's a lesson here. You can try to find methods and objects that make your life easier, but you still have to test. If they don't do what you want, you must figure out another way.

I added motion to the Left property of the player, recalculated motion using the same formula, and added it to the Top property. The net effect is that each element moves up to 5 pixels in each

direction randomly each time the timer ticks, making the nice random motion of players on the screen.

As you've heard by now, whenever you increment or decrement a variable, you should check for boundary conditions. In this case, this means making sure that the players don't move off the field. I did this by comparing the player's position with the field panel.

Moving the Opponents

The motion of the opponents is very much like that of the players. In fact, all I did was copy the code and change the picture box name to picOpp.

Hint Usually, when you find yourself copying code, it's an indicator that you should modify your data structure. I decided not to in this case because although the motion of the opponents and the players is identical, players and opponents are different in many other aspects, so they should still be different arrays.

```
private void moveOpp(){
  //move opponents
  int motion;
  Random roller = new Random();

  for (int i = 1; i < 5; i++){
    motion = roller.Next(11) - 5;
    picOpp[i].Left += motion;
    motion = roller.Next(11) -5;
    picOpp[i].Top += motion;

    //check for boundaries
    if (picOpp[i].Left < 0){
      picOpp[i].Left = 0;
    } else if (picOpp[i].Left +
picOpp[i].Width > pnlField.Width){
      picOpp[i].Left =
pnlField.Width - picOpp[i].Width;
    } else if (picOpp[i].Top < 0){
      picOpp[i].Top = 0;
    } else if (picOpp[i].Top +
picOpp[i].Height > pnlField.Height){
      picOpp[i].Top =
pnlField.Height - picOpp[i].Height;
    } // end if

  } // end for loop
} // end moveOpp
```

As in the player movement scheme, the position of the opponents has no bearing on the game play. In fact, the opposing players are completely unnecessary. They are just there for visual effect.

Updating the Score

Updating the score is an easy chore, but because it's done from several places, it is a good candidate for a method call:

```
private void updateScore(){
  lblPlScore.Text = "Player: " +
playerScore.ToString();
  lblOppScore.Text = "  Opp: " +
```

```
oppScore.ToString();
} // end updateScore
```

To update the score, all I had to do was convert the playerScore and oppScore values to strings and copy them to the appropriate labels.

Summary

This chapter helps you begin thinking like an experienced programmer. The way you work with complex data has a profound effect on how successfully you solve interesting programming problems. In this chapter you learned how to build arrays to store lists of information. You learned how to use the properties and methods of the array class to inspect and modify an array. You learned how to build arrays of simple elements such as integers and strings and more complex elements such as picture boxes. You learned how to build a form without using the Visual Designer and how to add event handlers to components you have built by hand. Finally, you put all these skills together to build an interesting game. In the next chapter you will move on to another important part of data manipulation: moving data to and from files.

Challenges

- Add multiple difficulty levels to the game. Let the user choose a difficulty, and adjust the opponent's likelihood of scoring based on the difficulty factor.
- Change the likelihood of a pass so that it is related to the distance between the shooter and the target. You might recall the distance formula from math class: distance = square root of ($(x_2 - x_1)^2 + (y_2 - y_1)^2$)
- Write a method that accepts two picture boxes, calculates the distance between them, and makes the likelihood of success dependent on that distance.
- Make a Windows version of the Snowball Fight program from Chapter 5, "Constructors, Inheritance, and Polymorphism: The Snowball Fight." Use picture boxes to indicate the players' behavior and buttons to get commands from the user.
- Modify the behavior and performance of the Soccer game to mirror a similar sport (hockey, basketball, and volleyball make interesting candidates). Change the images and behavior to mimic the sport. For example, basketball is a much higher scoring game than soccer, so the likelihood of scoring has to be much greater.

Chapter 9: File Handling: The Adventure Kit

You are beginning to see how important data is to a programmer. Often, even when your program is not running, you want the data in your program to be permanently stored so that you can retrieve it later. Modern languages such as C# support many tools for storing and retrieving information on disk drives. In this chapter you will learn two of the most basic and important schemes: text file management and object serialization. Text files are used to store basic data, and object serialization enables you to save and load entire objects. You will also learn how to use dialog boxes to streamline certain kinds of user interaction and how to add menus to your programs. In this chapter you will learn how to

- Store and load a plain text file to a disk.
- Use file streams to manage your interactions with disk files.
- Create multilevel menu systems on your forms.
- Use the File, Font, and Color dialog boxes to interact with the user.
- Store entire classes with object serialization.
- Write programs that use multiple forms.

Introducing the Adventure Kit

The Adventure Kit game program you will produce in this chapter is fun in a couple ways. For one, it allows you to play simple adventure games. However, the real appeal of the program is that it encourages the user to create original adventure games. One key to making a program interesting is to make it extensible. You can use the Adventure Kit to develop many types of adventure games.

Viewing the Main Screen

The Adventure Kit game uses several screens to control the action. The main screen is extremely simple. It allows the user to load, play, or edit (or create) a game or quit the program. This simplistic main program is featured in Figure 9.1. I deliberately made the main screen simple because the other parts of the program can be complex and I wanted to control the level of complexity. For example, a complete beginner to the program would probably want to play a game or two before calling up the editor.

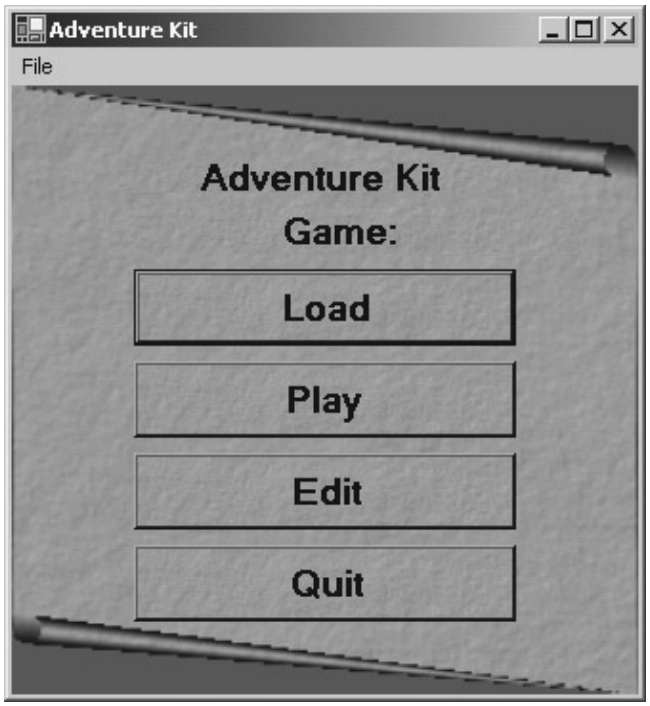


Figure 9.1: The main screen provides very basic access to the other parts of the program.

Loading an Adventure

The Adventure Kit game is designed to allow multiple adventures. Each adventure is stored as a file on the drive system. The adventure files can be shared between users. To play an adventure, you must load it into memory first.

When the user clicks the Load button, a standard file dialog appears (see Figure 9.2). The File dialog box uses very standard Windows metaphors to handle selecting a file from the drive system. It looks like a complex form to build, but don't panic. A trick is coming, in the section titled (strangely enough) "Using the File Dialog Controls."

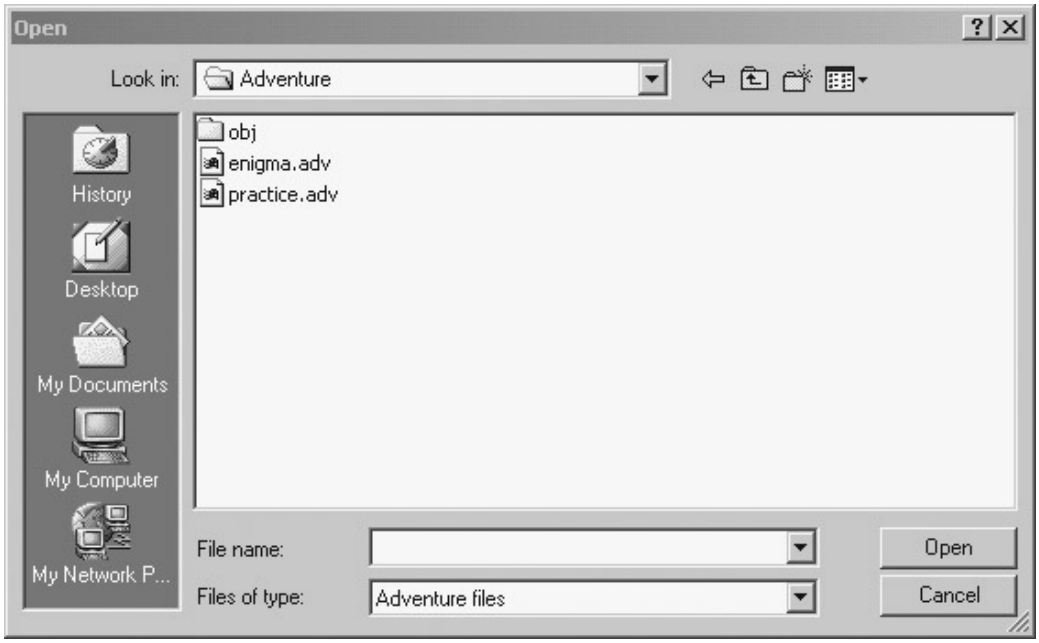


Figure 9.2: The file dialog looks very familiar to experienced users of Windows.

Playing an Adventure

Of course, the obvious thing to do with the Adventure Kit game is to play an adventure. If the user clicks the Play button, a second screen appears, with a game preloaded into it (see Figure 9.4). The user interface of the game program is designed to be as clean and obvious as possible so that the user can concentrate on the game rather than on the mechanics of moving around.

I wrote a simple adventure named *Enigma*. In this adventure, you are a spy sent to capture the famous enigma code machine from a Nazi submarine. Figure 9.3 through Figure 9.5 show a few scenes from this adventure. When the user is finished with the game, he can close the game form with a command from the menu. Alternatively, the player can edit the current game, also with a menu command. The menu system is nice because most users already know how to use menus.

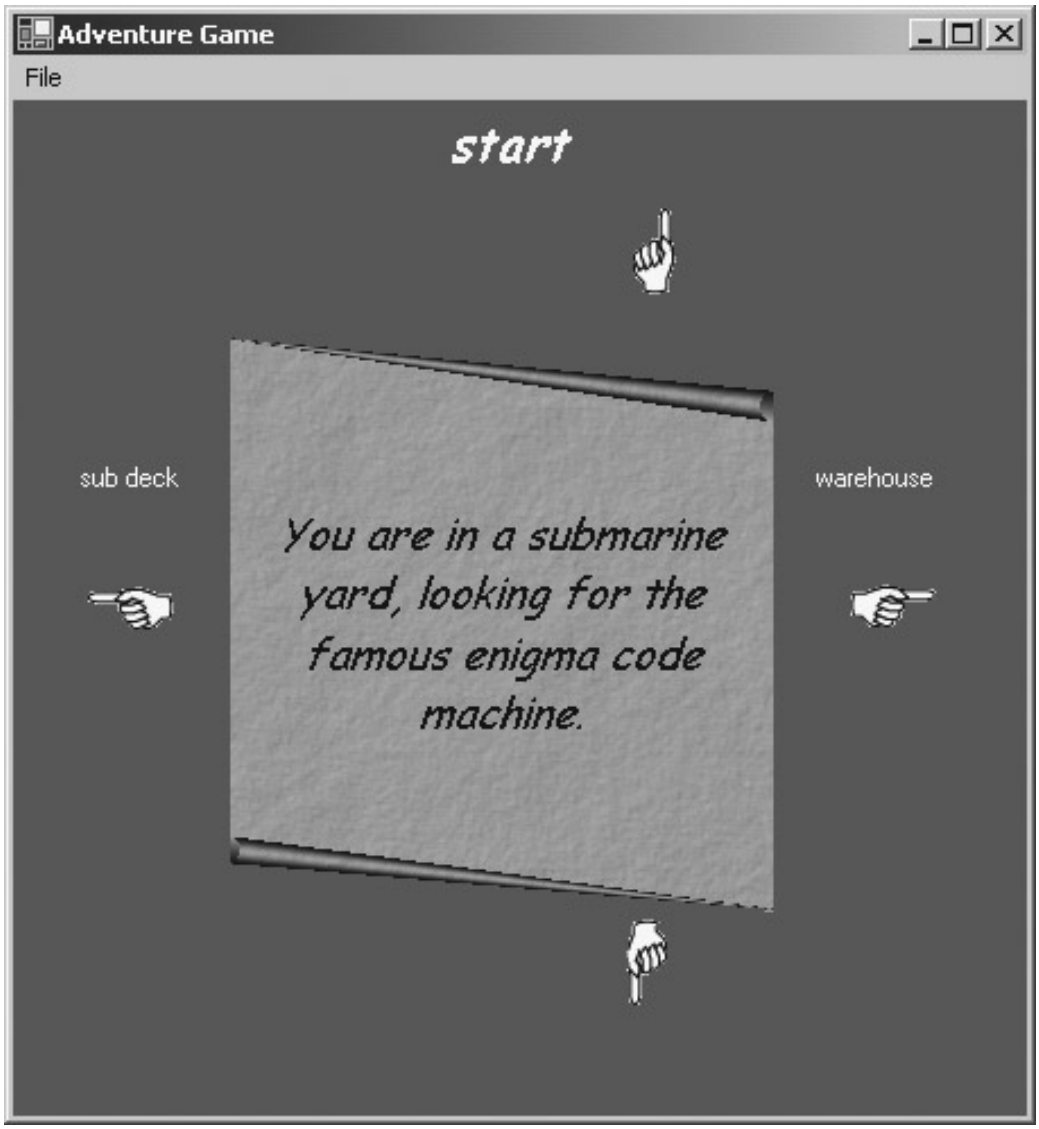


Figure 9.3: Here's the starting situation. Try clicking Sub Deck. (The Enigma device was carried on submarines.)

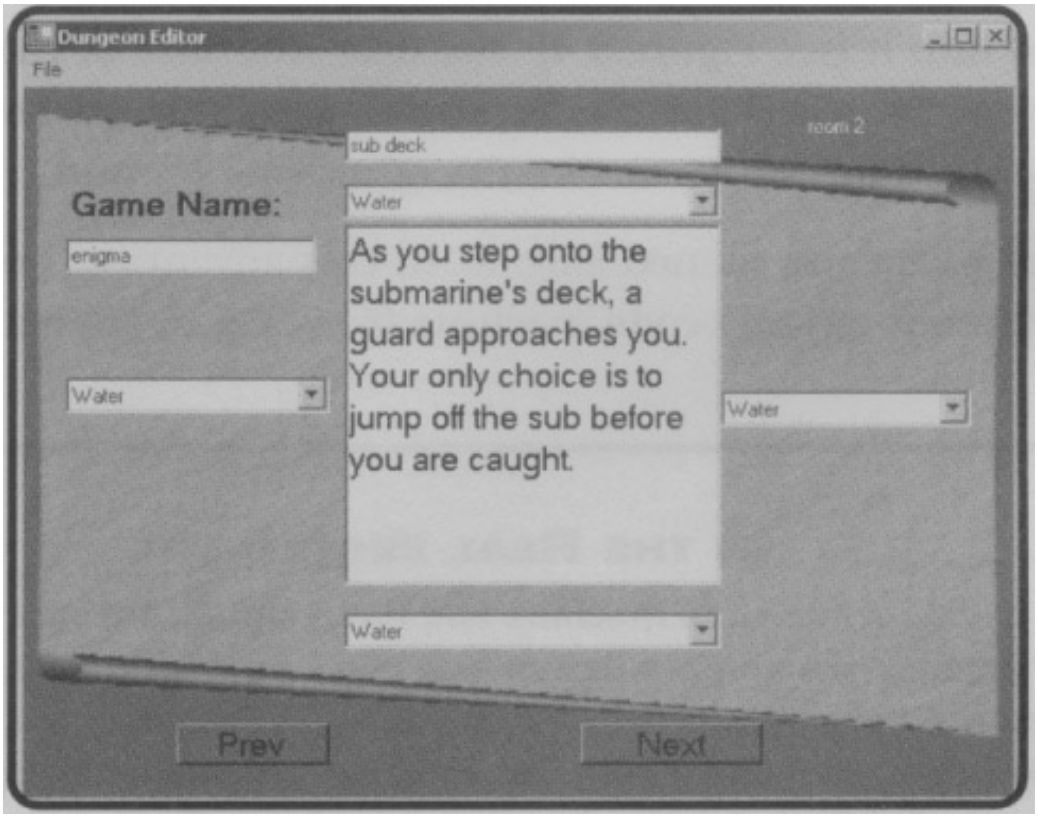


Figure 9.4: The screen changes to reflect the new situation. It's not looking good, but there are plenty of places to jump.

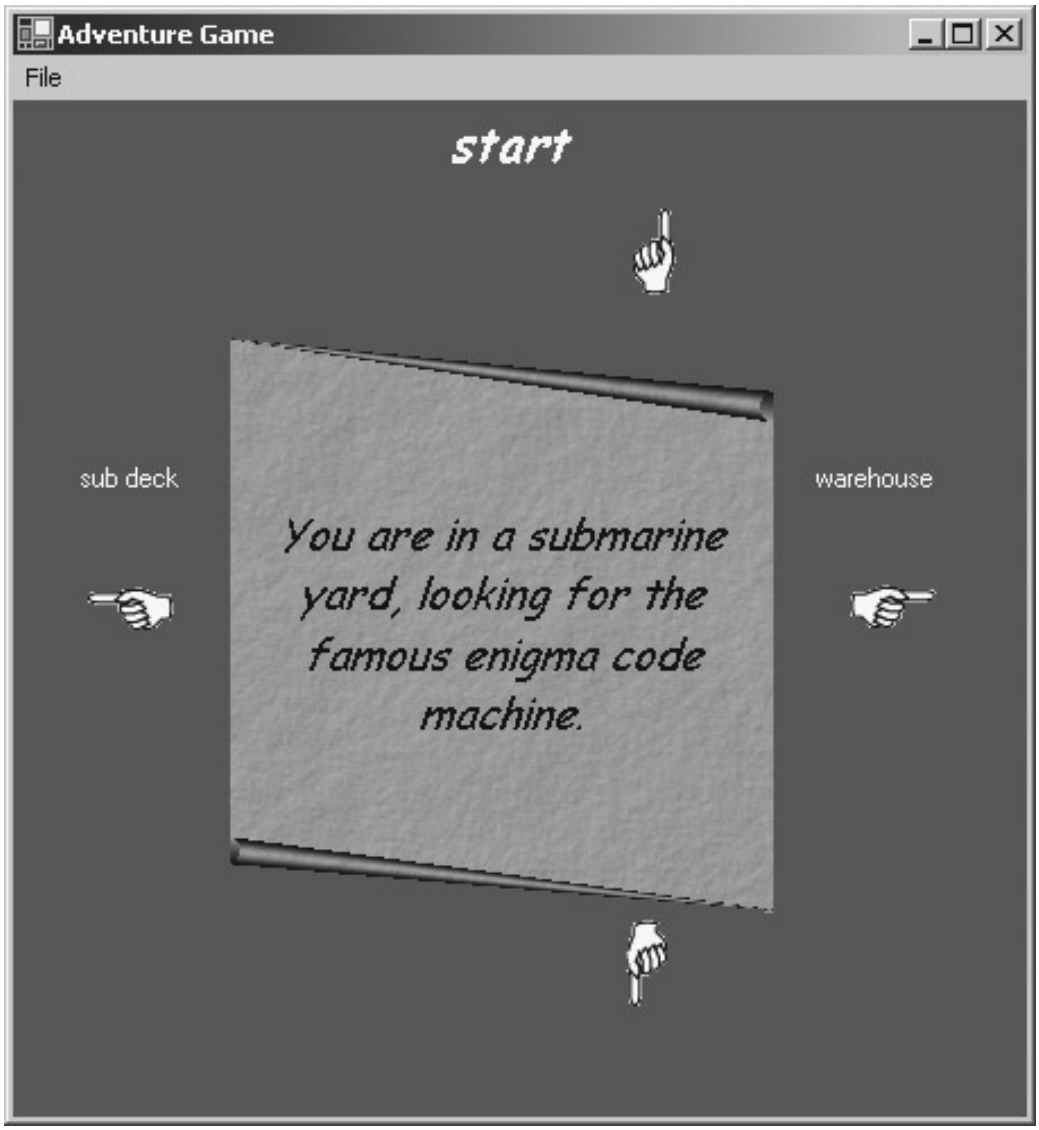


Figure 9.5: Oh, no!

In the Real World

The enigma device was a code machine and has a significant role in the history of computer science. An enigma device was recovered from a Nazi submarine, and a team of mathematicians led by Alan Turing built one of the earliest computers (named *Colossus*) specifically to crack the enigma code. In Turing's day, the notion that a computer (there were supposedly four computers in the world at that time, and all were top secret) could work on information besides numbers was considered extremely radical. It’s a fitting tribute to Turing that this sample program, related to his work, illustrates the fun and usefulness of nonmathematical data.

Creating an Adventure

When you finish saving the world, you can also use the Adventure Kit to build your own adventure games or modify existing games. Use the menu system to close the game window and return to the main screen. From there, call up the Dungeon Editor by clicking the Edit button. The editor screen is featured in Figure 9.6. It is similar in general design to the game screen but has some important differences.

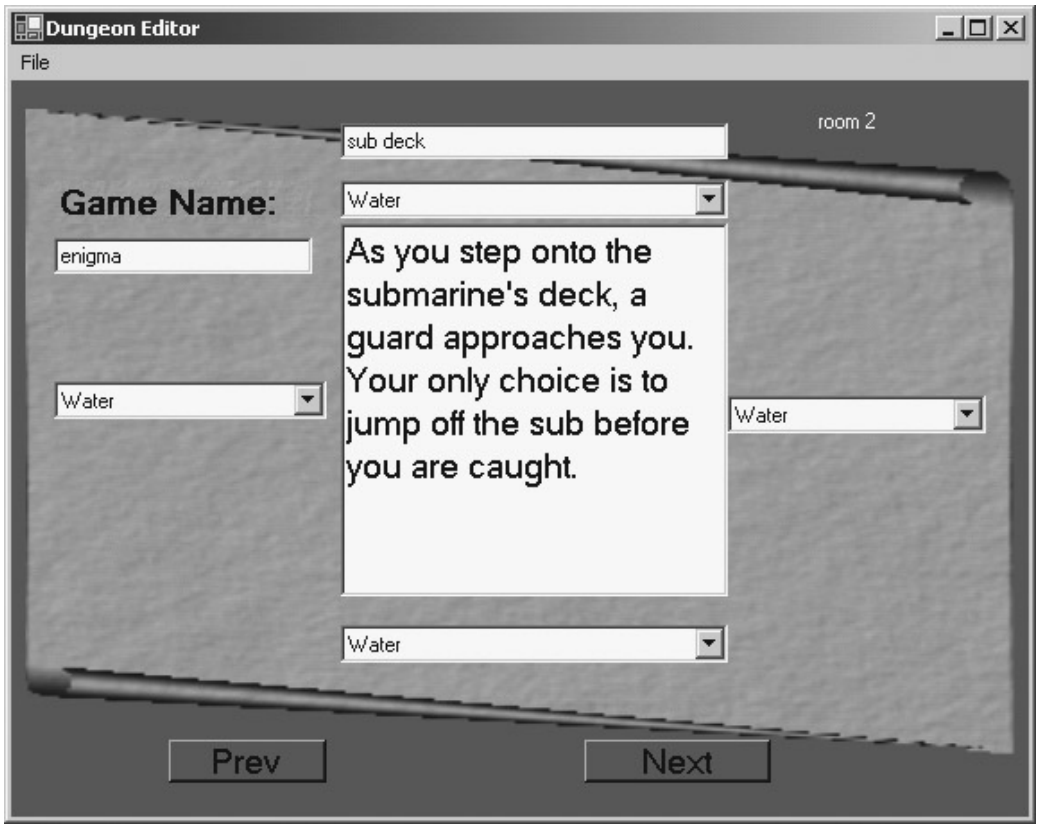


Figure 9.6: In the Dungeon Editor, you can change the values of the screen elements.

The editor is designed along the same general lines as the game program but is designed for building and editing adventures rather than playing them. Each game is designed as a series of rooms. A room has a name, a description, and links to as many as four other rooms. You can type the name of the room and its description directly into the appropriate text boxes and can choose the other rooms from the drop–down list boxes. You can move between rooms in the editor with the Next and Prev buttons.

Trick I've found that the easiest way to write an adventure game with this system is to start by entering all the room names first. When I come back to edit the rooms, all the other room names are available in the list boxes. Creating all the connections then becomes a simple process. Finally, I flesh out the story by adding the descriptive text.

That the editor and the game engine use a similar interface is handy because it speeds up the learning curve. You can safely assume that a user interested in creating or editing an adventure has already played at least one, so having the elements in familiar places makes learning the editor easy for a user.

Reading and Writing Text Files

Before you can write a program as involved as the Adventure Kit, you need to understand how computers work with files. Information in variables is very powerful, but variables are essentially references to memory. Most computer memory is *volatile*, which means that it can store values only while power is flowing through the computer. If you want to store values that live beyond one session at the computer, you need to be able to store information in files on a disk drive. Perhaps the simplest kinds of files are *plain text files*, which store information in pure ASCII form on the disk. These files are easy to work with, and they are universal. Nearly every computer made can store and load text files. This is why most Internet standards are based on the plain text format.

C# uses the concept of *streams* to manage files. Essentially, a file can be thought of as a continuous stream of data. Some streams specialize in providing data to the program. These are known as *input streams*. Other streams are used to store information to a drive system. These *output streams* specialize in sending data from the program to the drive system.

Exploring the File IO Program

The File IO program, displayed in Figure 9.7, is an example of simple input and output (collectively called *IO*) using input and output streams. The user interface of File IO has one new feature. So far in this book, most text boxes have displayed only one line of information. It's possible to set up a text box to show multiple lines of data, using the Multiline property. This is useful when you want to display more than one line in a textbox. In the FileIO program, I show the entire file in a textbox, so the Multiline property makes the text much easier to read.

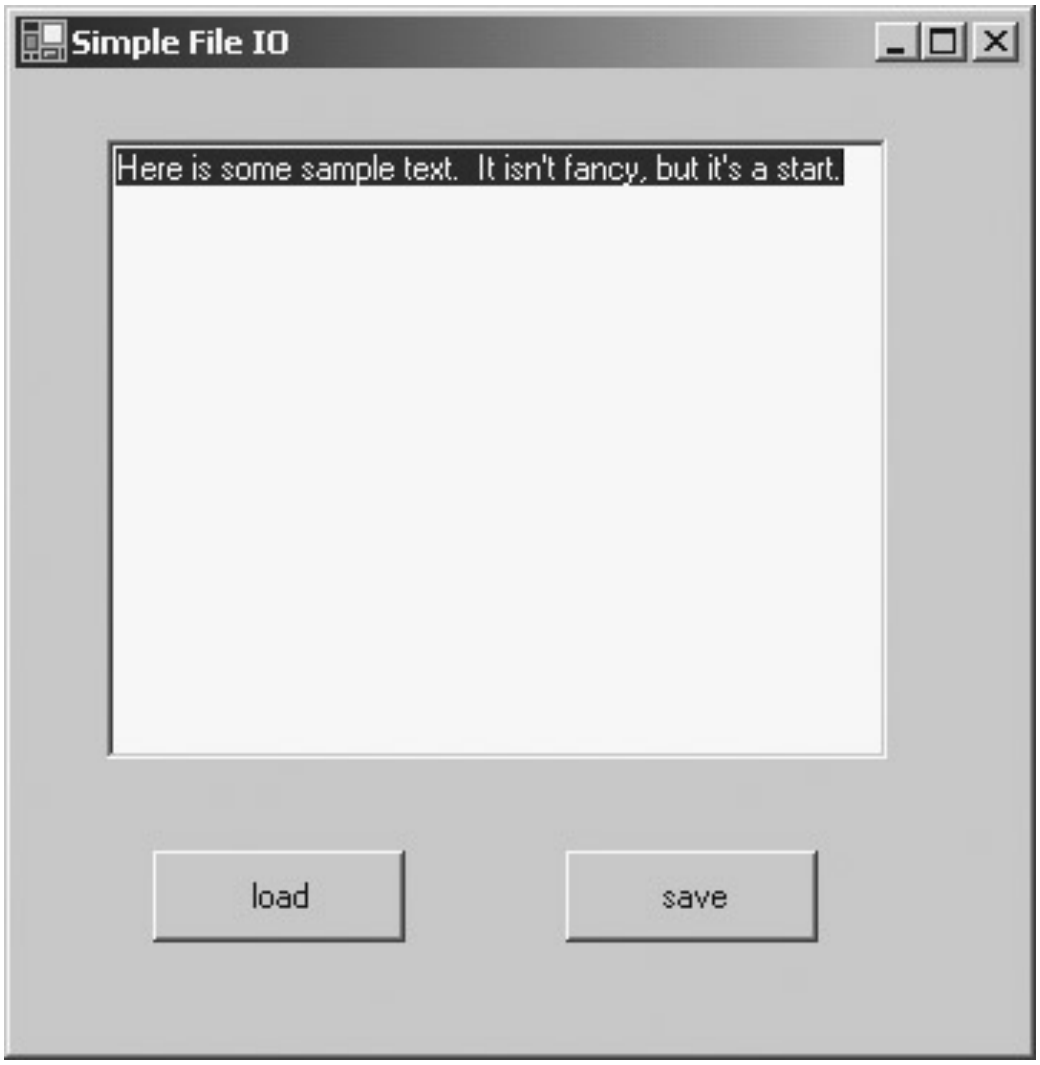


Figure 9.7: The File IO program has a large text box and buttons for saving and loading the file.

Importing the IO Namespace

The input and output features of C# are stored in a namespace called *System.IO*. Figure 9.8 shows the help system's entry point into the System.IO namespace. The System.IO namespace has very useful classes. Table 9.1 describes a few of the more important classes in this namespace.

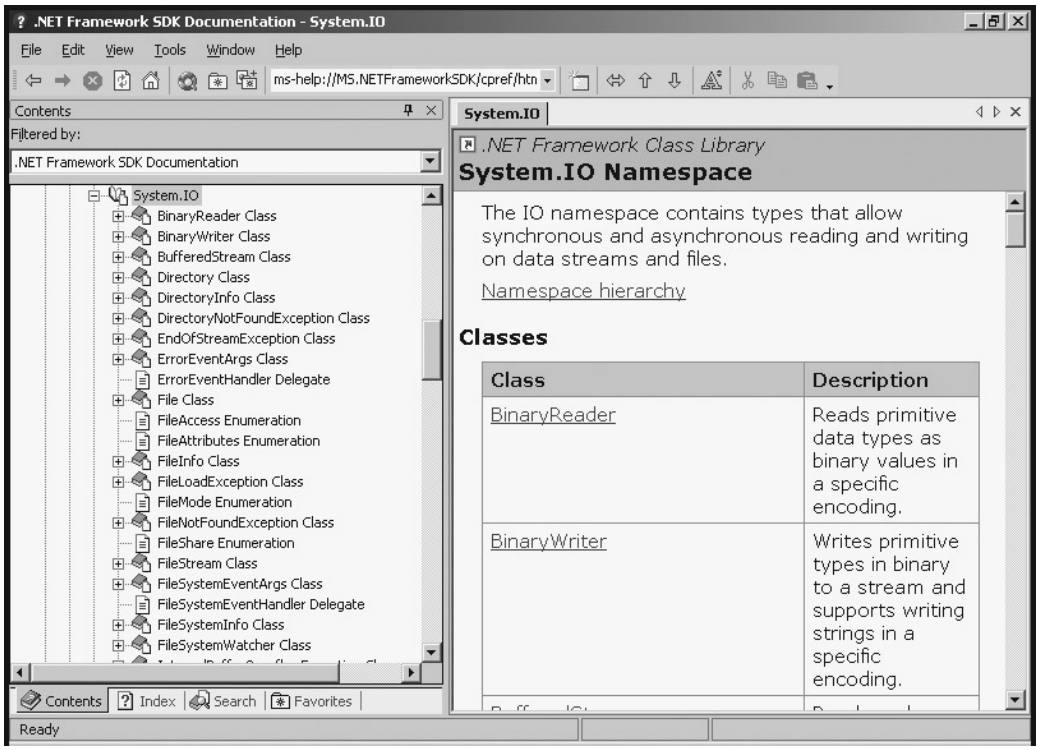


Figure 9.8: The System.IO namespace includes classes for input and output streams.

Table 9.1: Significant Classes in the System.IO Namespace

Class	Description
File	Features several methods for moving, copying, and testing a file (to see whether it exists, when it was created, what kind of access it allows)
FileStream	The basic file class, allows the author to open, close, read, and write to files, generally used for binary or random access data
StreamReader	A descendant of the FileStream class optimized for reading text–based information from a file
StreamWriter	A descendant of the FileStream class optimized for writing text–based information to a file
Path	Allows the programmer to create a valid file path from string data
Directory	Allows the programmer to create and manipulate directories

Many other classes are available in the System.IO namespace, but the classes in Table 9.1 are a good starting point.

Any of your programs that will use File IO must import the System.IO namespace. The line looks like this:

```
using System.IO;
```

Trick I didn't show you the rest of the code for the startup of the File IO program because, except for the event handlers, it is the default code generated by the designer. You can see the program in its entirety on the CD–ROM.

Writing to a Stream

The interesting parts of the File IO program happen in the event handlers of the two buttons. The Save button has code for writing the contents of the text box to a file stream:

```
private void btnSave_Click(object sender, System.EventArgs e) {
  StreamWriter sw = new StreamWriter("practice.txt");
  sw.Write(txtEdit.Text);
  sw.Close();
} // end btnSave
```

The btnSave() method is an event handler, which is automatically called when the user clicks btnSave. The method begins by creating an instance of the StreamWriter class. StreamWriter is optimized to write text files. It has a number of constructors, but the easiest requires simply the name of the file you want to write. The method then uses the Write() method of sw to write the text from the text box to the file. The last line of the method uses the Close() method to close access to the file.

Trap Forgetting to close files is common. If you make this mistake, the program will continue to run, but you will encounter a file access error the next time you try to open the same file—because it's already open. This can be an ugly bug to track down because your program crashes not *where* the error occurs but *the next time* the program tries to access the file. Be sure to close every file after you're done accessing it.

Writing to a text file in this manner is extremely simple. If you want, you can also write data one line at a time to the file. The StreamWriter class supports many of the same methods as the Console class—such as Write() and WriteLine()—so it isn't difficult to use.

Reading from a Stream

Reading text from a file is much like writing to a file, except that you create an instance of the StreamReader class instead of the StreamWriter class. The code for btnLoad() in File IO demonstrates how to read the text from the file:

```
private void btnLoad_Click(object sender, System.EventArgs e) {
  StreamReader sr = new StreamReader("practice.txt");
  txtEdit.Text = sr.ReadToEnd();
  sr.Close();
} // end btnLoad
```

As you can see from the code listing, the StreamReader class is very easy to use. To read data from the stream, you use reading methods. The ReadLine() method works just like the ReadLine() method in the Console class. It reads from the stream until it encounters an end–of–line character. Often, you want to read the entire text of a file into a variable. The ReadToEnd() method quickly reads the entire text file.

In the File IO program, I decided to predetermine a file name. Letting the user choose a file name would be better, but this can be a dangerous exercise because you never know what the user will type. In the Dialog Demo program later in this chapter, I'll show you how to let the user choose a file. Meanwhile, turn your attention to menus.

In the Real World

The text methods outlined here work fine for small text files without a lot of fancy formatting. These methods become more cumbersome when you are working with a large amount of data or when the data has very specific organization. In those situations, you might want more specialized storage techniques, such as the object serialization strategy described in the Serialization Demo program later in this chapter, the XML strategies described in Chapter 10, "Basic XML: The Quiz Maker," or the database strategies described in Chapter 11, "Databases and ADO.NET: The Spy Database."

However, all those techniques share the basic file manipulation ideas presented here as their genesis.

Creating Menus

Menus have become a staple GUI component. Programmers like menus because they allow access to large numbers of commands and methods without requiring much space on the screen. Users like menus because they are predictable and allow access to much of the program's functionality (if they are written well, that is!).

The .NET IDE makes building a menu structure for your programs very easy. Menus are basically composed of two classes. The MainMenu class can be dropped on your form, and it provides the basic foundation of your menu system. The user doesn't interact with the MainMenu class itself. Instead, the MainMenu has a series of MenuItem components. These MenuItems are the parts of a menu that are visible to the user.

Exploring the Menu Demo Program

The Menu Demo program consists of a simple form using GUI menus. Figures 9.9 and 9.10 show the completed program. Because users already know how to use a menu, you don't have to teach this skill. However, it's up to you to design your menus so that they are familiar to users. For example, most programs have a File menu, followed by an Edit menu. Users expect to find saving and loading commands on the File menu and copying commands on the Edit menu. If you don't follow these conventions, expect some confusion from your users.

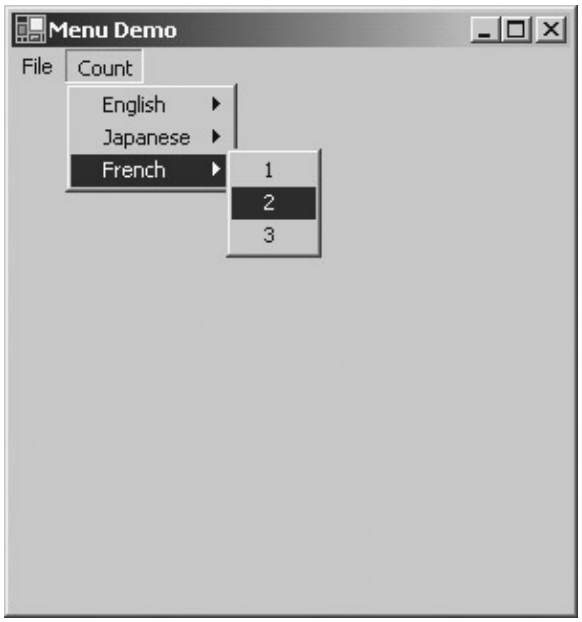


Figure 9.9: The form has one main menu, which has menu items. The menu items have submenus.

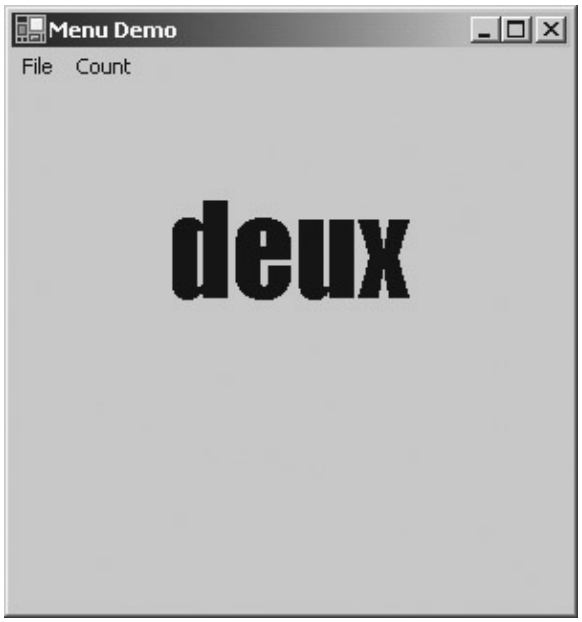


Figure 9.10: After the user makes a selection, the program prints out the number name.

Adding a MainMenu Object

Adding a menu to a GUI form is easy. Select the MainMenu control from the Toolbox on the left of the IDE screen, and drag it onto the form. The MainMenu control places itself in the area immediately below the form (where the timer and the image list go). You won't see the MainMenu control on the form, but if you look in the upper–left corner, you will see that a MenuItem has been placed on the form for you. Figure 9.11 shows a form with one menu being placed on it.

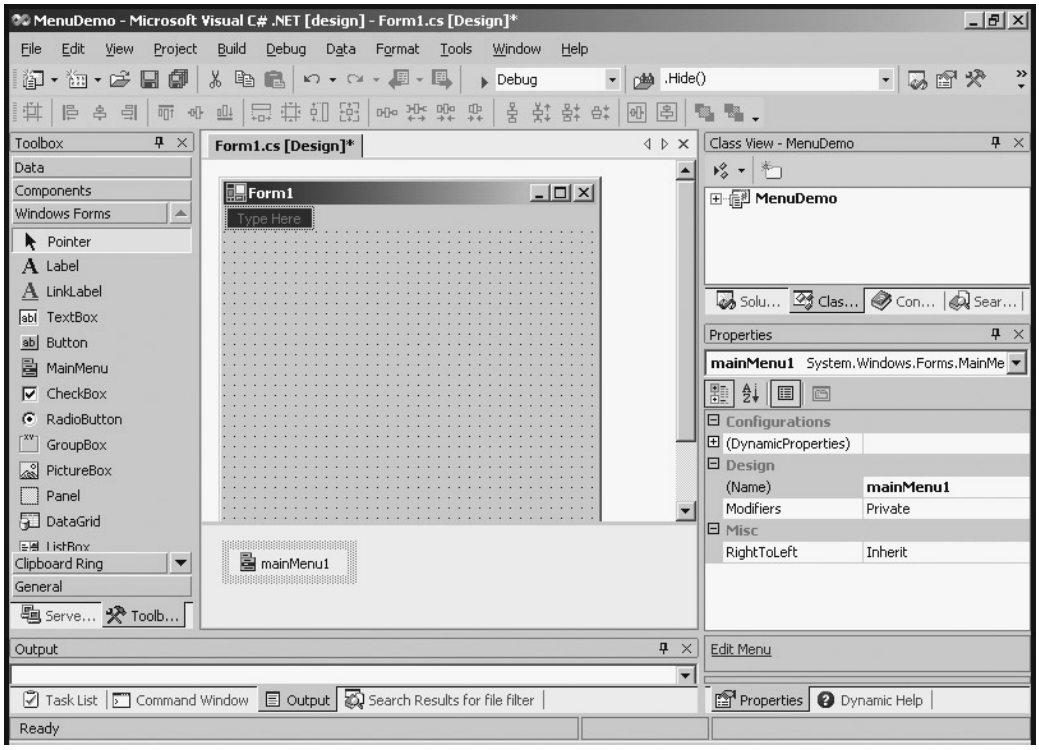


Figure 9.11: The MainMenu object is below the form, and the first MenuItem object has been placed at the top of the window. (It says Type Here.)

It's important to note that MainMenu and MenuItem are two different objects. MainMenu is used to attach menus to your form. After you add MainMenu, you don't need to think much about it. The MenuItem class is used to generate all the items on the menu. The menus themselves (such as the

File menu and Edit menu) are MenuItems. Every element in the menu is also a MenuItem.

Adding a Submenu

To make a submenu, type in the Type Here box. I want to re–create the Menu Demo program shown at the beginning of the chapter, which has a File menu and a Count menu. The leftmost menu is File, so you type the word **File** in the Type Here box. Figure 9.12 shows what happens when you start to type in the box.

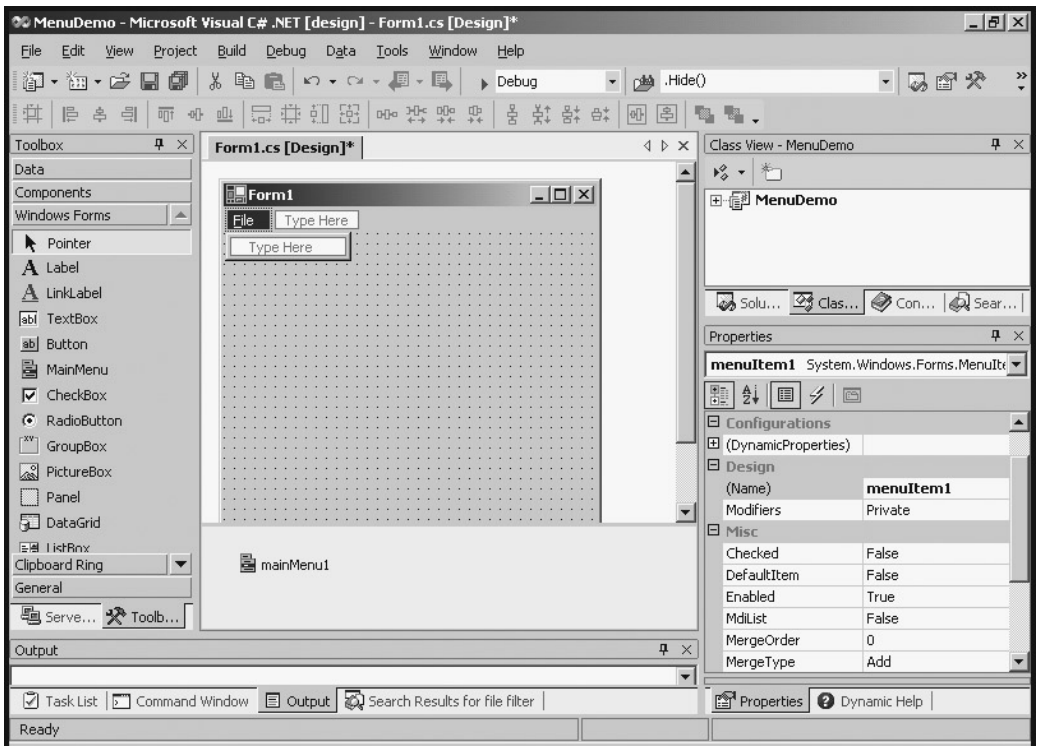


Figure 9.12: When you start to type in a menu item, two new potential menu items appear!
As soon as you begin to type code in a menu item, two new Type Here boxes appear. These are placeholders for future menu items. When you finish typing

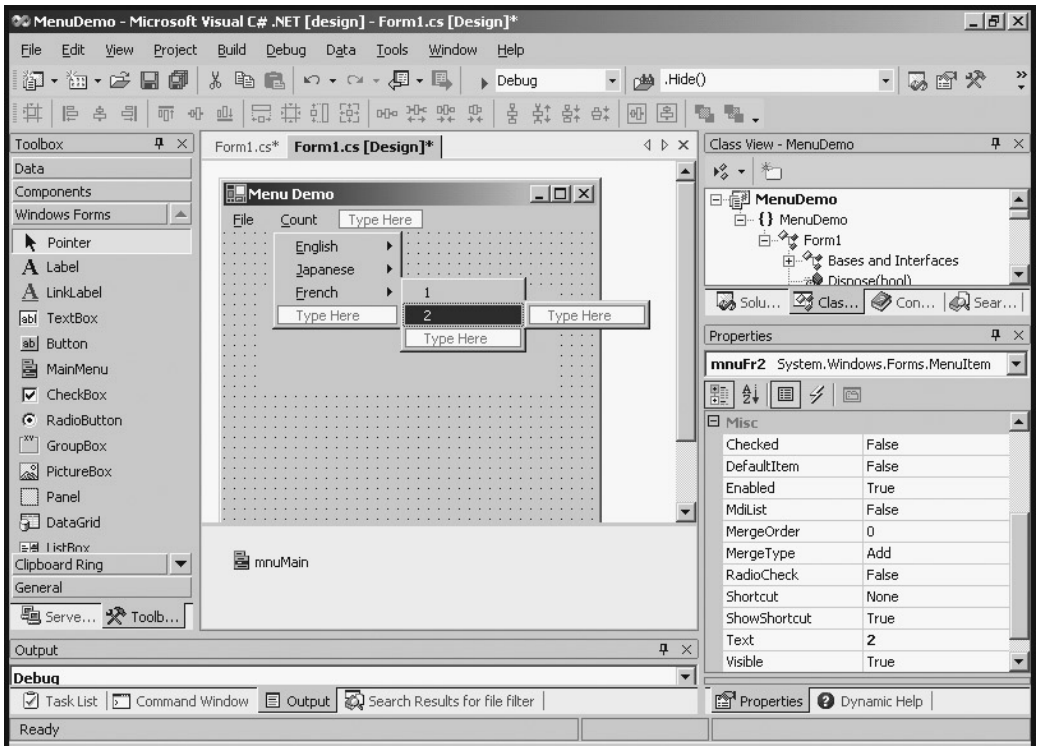


Figure 9.13: Every element opens up the possibility for two more elements.

File in the upper–left box, you add **Exit** below it and **Count** to the left. Figure 9.13 shows what happens after I added a few more elements.

Trick You can create very complicated menu structures, if you like. Keep in mind, though, psychologists' observations that most people can handle only seven options at a time. Subdividing your menus into reasonable chunks is smart, but menu structures 12 layers deep can be frustrating. As an example, look at the menu structure for Visual Studio itself. You can use the program for years and still not know where everything is on that behemoth. (In fact, you can even change the menu structure if you want, adding new custom commands and deleting things willy–nilly. This is a setup for a great April Fool's joke at the office but makes the interface even more confusing than it already is.) The point of a menu structure is to make your program easy to use. Keep your menus simple and direct.

Setting Up the Properties of Menu Items

Menu items are objects, and they have properties, just like text boxes, labels, and all the other elements you place on forms. Like those other elements, the MenuItem class has properties that add functionality to the method. The most important property to change in a menu item is the name. The default names for menu items are not clear, and you usually have many of them if you have any at all. Your code will soon become very confusing if you start to write event code for the menu items without changing their names.

Trap With all controls that will have event handlers, it's critical to change the object's name *before* you assign an event handler to it (by double–clicking the element or choosing the event from the event list). If you assign an event and then change the object's name, the event will still occur, but the event handler will refer to the old object name, not the new one. With menu events, this is even more critical than with other kinds of events because all the menus have very similar default names and there are often *many* menu items in a program.

In addition to the Name property, you can assign a check mark to a menu item. This is often used if you want to indicate that something can be turned on or off, such as italic. You can also assign a shortcut such as a function key or control key combination to a menu item, and when the user assigns the shortcut, any code associated with that menu is triggered. This is a very easy way to add keyboard commands to your programs without having to call a keyboard handler explicitly. The Shortcut property of the MenuItem object has a drop−down list of legal keystroke commands you can associate with a menu item.

Menu item names often have one character underlined. If you press the Alt key and then the underlined letter (sometimes called a *hotkey*), control immediately jumps to the menu item. For example, many users save documents with the Alt+F+S combination. Alt+F calls the File menu in many programs, and the S key usually invokes the Save command on the File menu. To add an underscore (and the associated behavior) to a menu item, precede the character you want to underline with an ampersand. For example, change *File* to &*File* to underline the F key and have it act as the hotkey for the File menu.

Trick The system of hotkeys also works with command buttons. It can be an easy way to support those users who prefer to use the keyboard. Also, keyboard shortcuts such as the hotkey combinations can greatly simplify life for users with visual impairments or other disabilities, who find keyboard commands much easier to manage than mouse−based input. In any case, hotkey commands add a lot to your menu structure and are very easy to add to your programs.

Writing Event Code for Menus

Menus are almost always associated with some sort of event handling (except for the topmost menus, such as File and Count, although you could add some sort of event to them, if you wanted to). You add event handlers to a menu item just like any other control. Double−click the menu item to call up the default event handler, or choose another event from the Events list on the Properties window.

The following listing shows the event handlers for the Menu Demo program:

```
private void muEng1_Click(object sender, System.EventArgs e) {
  lblNumber.Text = "one";
}

private void mnuEng2_Click(object sender, System.EventArgs e) {
  lblNumber.Text = "two";
}

private void mnuEng3_Click(object sender, System.EventArgs e) {
  lblNumber.Text = "three";
}

private void mnuJapan1_Click(object sender, System.EventArgs e) {
  lblNumber.Text = "ichi";
}

private void mnuJapan2_Click(object sender, System.EventArgs e) {
  lblNumber.Text = "ni";
}

private void mnuJapan3_Click(object sender, System.EventArgs e) {
  lblNumber.Text = "san";
}
```

```
private void mnuFr1_Click(object sender, System.EventArgs e) {
  lblNumber.Text = "un";
}

private void mnuFr2_Click(object sender, System.EventArgs e) {
  lblNumber.Text = "deux";
}

private void mnuFr3_Click(object sender, System.EventArgs e) {
  lblNumber.Text = "trois";
}

private void mnuExit_Click(object sender, System.EventArgs e) {
  Application.Exit();
} // end setupMenu
```

Using Dialog Boxes to Enhance Your Programs

As programmers started developing GUI software, it quickly became apparent that certain kinds of complex tasks are needed again and again. For example, allowing the user to choose a file is a potentially risky operation because the programmer doesn't know anything about the user's file structure (how many drives are on the computer, what the drive names are, and so on). Also, allowing the user to type in a file name is simple to program but leads to many errors. Soon standards began to appear. If you think about most programs you use, a standard form appears when you start to save or load a program. This form takes control of the program until the user makes a choice. Forms like this are called *dialog boxes* because they facilitate a dialog with the user. As a user, the dialog box that appears when you save or load files seems to be much the same no matter what program you are using. This leads to shorter learning time (because the dialog box is already familiar to you) and fewer mistakes (because you already know how to get around in the dialog box). In the early days of GUI programming, developers had to be able to build file dialog boxes by hand. The same thing was true of color–choosing dialogs, print dialogs, and font selection dialogs. These dialog boxes were tedious to create by hand, and ensuring that they work exactly as users expect was challenging.

Fortunately, the .NET framework offers prebuilt, standard dialog boxes to help you include the more common dialogs used by programmers. Each of these dialogs is a class with properties, methods, and events. You can use the dialogs to greatly simplify your interactions with the user.

Exploring the Dialog Demo Program

The Dialog Demo program extends the text editor built in the File IO program. It replaces the command buttons with menus and adds access to several standard dialog boxes to change the program's appearance and behavior. Figure 9.14 shows the Dialog Demo.

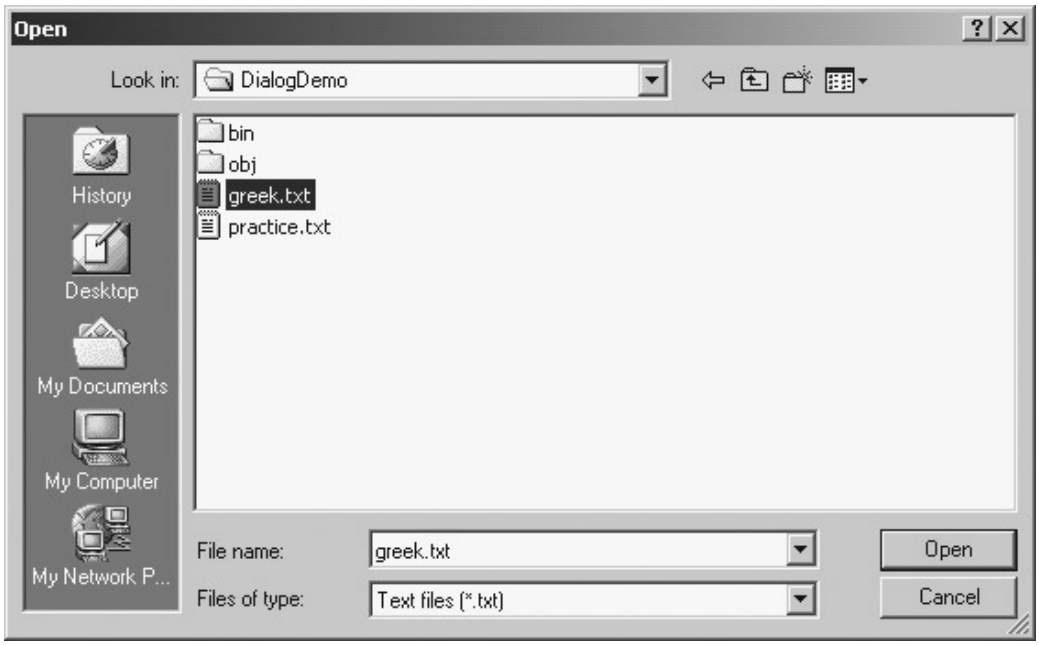


Figure 9.14: When the user chooses Open from the File menu, a familiar–looking dialog appears.

The file dialog makes your job as a programmer much easier because you don't have to worry much about how it works (isn't encapsulation grand?). All you have to do is drop it on the form, set a few properties, and respond to its events.

Notice that the dialog box is preset to display only text files (the file names end with *.txt*). You can set up a filter to display any kind of file you want. You can also determine a default directory, a default file name, and other interesting characteristics.

You can also change the font visible in the text editor with another custom editor. Figure 9.15 shows the font dialog.

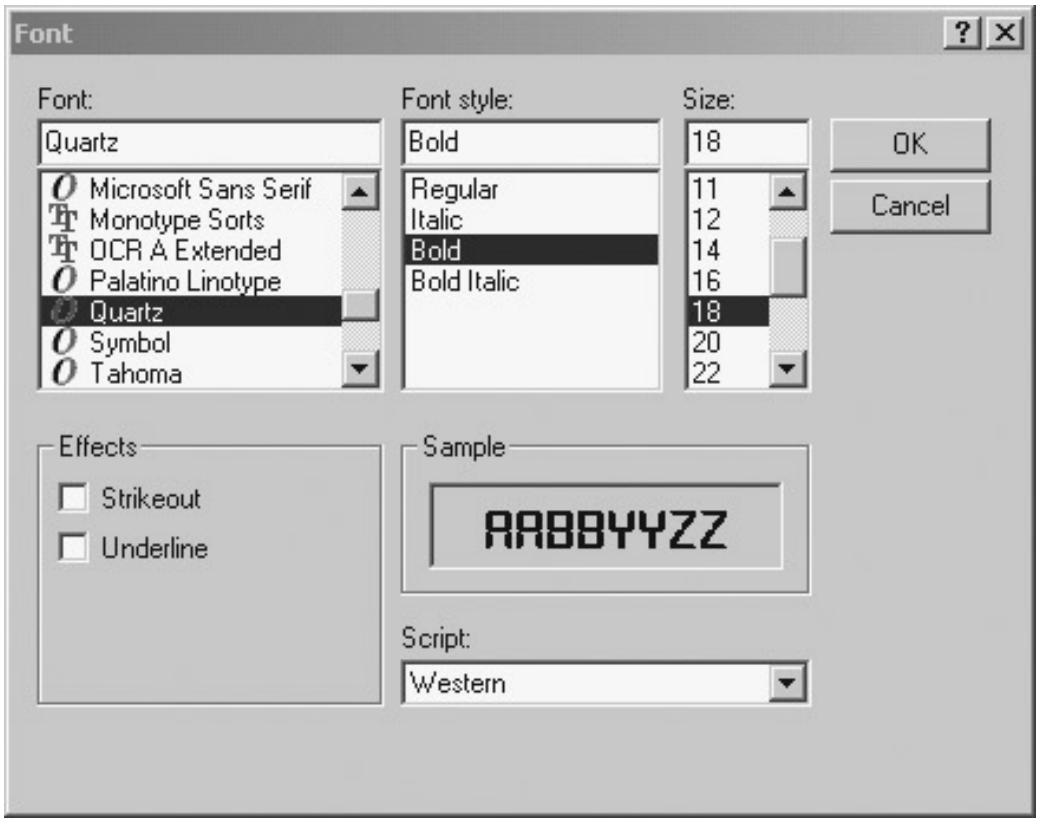


Figure 9.15: The font dialog allows the user to set the font, including typeface, size, and style.

The font dialog is an especially welcome tool because fonts have a long history of causing trouble for programmers. Each user can install whatever fonts he or she wants on his own computer. However, the programmer has no way of knowing directly which fonts are on the client machines because each user's machine has a different set of fonts installed. The font dialog sidesteps this problem by letting the user choose directly from the fonts installed on the user's computer.

The Dialog Demo program also allows the user to change the foreground and background colors of the text editor. Figure 9.16 shows this process. Like files and fonts, color can be a challenging attribute for the programmer to control. The color dialog gives a simple mechanism for allowing the user to choose a color. The color dialog is more sophisticated than it seems at first glance. The Custom Colors option allows the user to generate a color using RGB and HSL (hue, saturation, luminance) models.

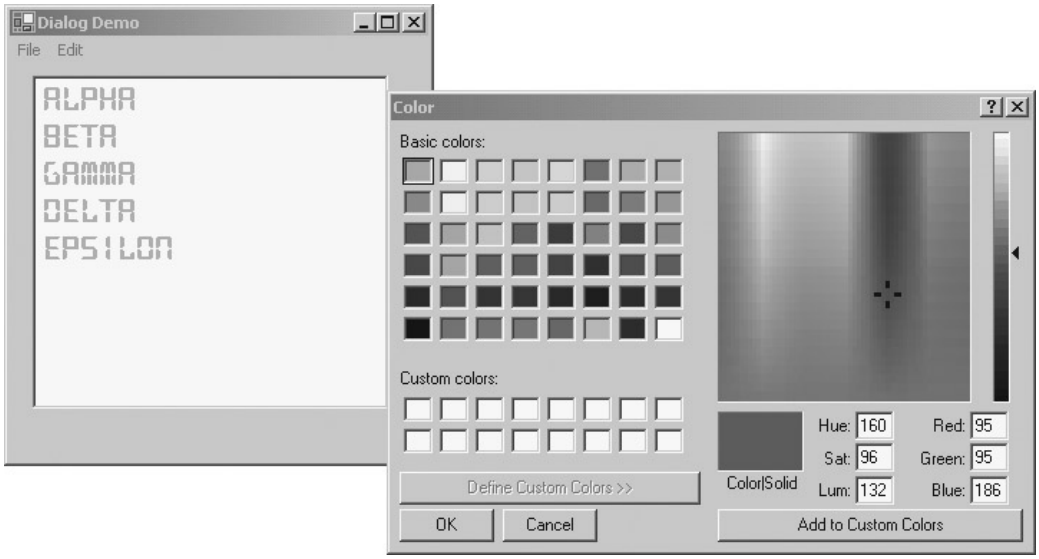


Figure 9.16: The user can independently set the foreground and background colors of the text editor.

Adding Standard Dialogs to Your Form

Although the standard dialogs seem to be a very diverse group of objects, they all work the same way from the programmer's point of view. Each dialog is a control you can drop on the form. All are available in the Toolbox to the left of the IDE window. As you drop the dialogs on your form, you see them added in the off–screen space at the bottom of the form, where timers and image lists go. Like these other controls, the dialog boxes don't have a physical presence on the screen; they appear at opportune moments under program control. Figure 9.17 shows the text editor with the dialog boxes added.

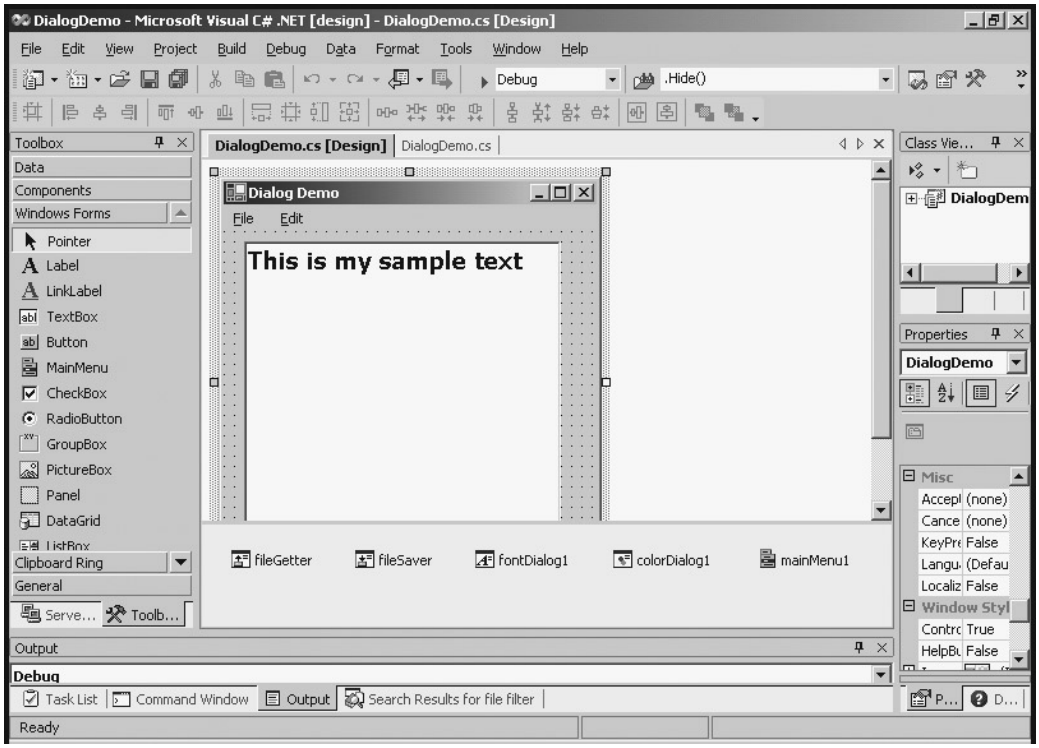


Figure 9.17: The dialogs are added to the bottom of the form. Notice the menu structure as well.

Trick The dialog controls usually appear towards the bottom of the Toolbox list, so they aren't immediately apparent in the IDE. You have to scroll down the Toolbox to see the dialog controls.

Like any other control, a dialog has properties that affect its appearance and behavior. Also, each dialog has at least one event it can trigger. The properties and events are associated with specific types of dialogs, so they are described separately in the sections that follow.

Using the File Dialog Controls

The OpenFileDialog and SaveFileDialog controls are almost identical. It's important to understand that the dialogs *don't actually save and load files!* As a user, you expect these dialogs to save and load files, but that isn't the case from the programmer's point of view. The real purpose of the file dialogs is to let the user choose a file name. It's up to the programmer to save or load the file. Although this might seem like a burden, it isn't, because the hardest part of saving and loading files is usually getting the file name from the user. Many other tools (such as the StreamReader and StreamWriter classes you learned about earlier in this chapter) help with the actual file manipulation. Dialogs are all about communication with the user. The file dialog controls have properties that simplify and direct this communication.

InitialDirectory

The InitialDirectory property lets you determine where in the user's directory structure the dialog will display at first. You can set this to whatever you want, but it's possible that the user's directory structure will be unknown to you. The safest alternatives are in a directory your program has created (using the System.IO.Directory class) or in the program's current working directory (denoted with a single period [.]). The default directory is wherever the program's binary files are installed on the user's machine. If you will be saving many files on the user's machine, you might want to make a directory off the default directory and instruct dialog boxes to open in this directory.

The initial directory is just a starting place. The user is still free to navigate the entire drive system (and perhaps even the network) to save and load files.

Trick Initial directories are especially important if you are writing programs for beginning users. Beginners often ignore the entire directory structure and save their files wherever the dialog first points them. This can lead to disaster if your load dialog doesn't point to the same initial directory as the save dialog.

Filter

The Filter property allows the programmer to determine which files will appear by default. The filter is useful because it limits the displayed files to those types of files the program is expecting. Typically, you either use a recognized standard file extension (such as *.txt* for plain text) or create your own extension for special–purpose file types. In either case, you should also add a filter to allow for all files, because often users still want to see every file in the directory. The file filter consists of a file description followed by the pipe symbol (|) and a pattern that returns the description. For example, this pattern

```
Text files (*.txt)|*.txt
```

places the value Text files (*.txt) in the Files of Type list box and displays all files that end with *.txt* in the directory listing. A filter can consist of multiple patterns. For example, both the DialogDemo file dialogs use the following pattern:

```
Text files (*.txt)|*.txt|All files (*.*)|*.*
```

Trap Be very careful with the patterns. If you put a space immediately after the pipe symbol, like this, Text files (*.txt)| *.txt the program will think that you are looking for files that begin with a space and end in *.txt*. Likewise, trailing spaces in a filter can cause the same kinds of problems. Test your filters carefully to ensure that they display exactly the types of files you're expecting.

Responding to File Dialog Events

The file dialog objects generate events just like most other components. The FileOK event occurs when the user clicks the OK button after choosing a file. You can write code in the default event handler to deal with saving or loading the file. To retrieve the file name requested by the user, use the Filename property of the OpenFileDialog or SaveFileDialog control (whichever one you were using). Here's the event handler code relating to the file dialogs in the Dialog Demo program:

```
private void mnuSaveAs_Click(object sender, System.EventArgs e) {
  fileSaver.ShowDialog();
}
```

```
private void fileSaver_FileOk(object sender,
        System.ComponentModel.CancelEventArgs e) {
  StreamWriter sw = new StreamWriter(fileSaver.FileName);
  sw.Write(txtEdit.Text);
  sw.Close();
}

private void mnuOpen_Click(object sender, System.EventArgs e) {
  fileGetter.ShowDialog();
}

private void fileGetter_FileOk(object sender,
        System.ComponentModel.CancelEventArgs e) {
  StreamReader sr = new StreamReader(fileGetter.FileName);
  txtEdit.Text = sr.ReadToEnd();
  sr.Close();
}
```

The mnuSaveAs_Click() method shows the fileSaver dialog box with the ShowDialog() method. The dialog then takes control of program flow until the user clicks OK or Cancel. If the user chooses Cancel, nothing further happens and the dialog closes down. If the user chooses OK, the fileSaver_FileOK() method leaps into action. It generates a stream writer based on the user's choice of a file name. The user's file name is stored in the FileName property of fileSaver, which is a SaveFileDialog. The method then writes the text to the stream writer and closes the stream.

The code for opening a file is very similar. A menu item click event—in this case, mnuOpen_Click()—shows the fileGetter, which is an OpenFileDialog. The fileGetter_FileOK() method triggers when the user clicks the OK button. This causes the creation of a StreamReader based on the user's choice. The program reads the contents of the stream and copies the data to the text box. Finally, the method closes the StreamReader.

Using the Font Dialog Control

The FontDialog control works much like the file dialogs. It doesn't change a font but asks the user to interactively choose a font, and that font is available as a property of the dialog for the programmer's use. You add a font dialog to a form in the same way you add a file dialog. Drag it onto the form, and it moves to the off–screen area. Generally, it isn't necessary to modify any of the font dialog's properties, but you can preset the font to a font that will work well for your program. Although font dialogs also generate events, it makes more sense to work with the font information in the same code that calls the dialog. The code for the font menu item illustrates how this can be done:

```
private void mnuFont_Click(object sender, System.EventArgs e) {
  if (fontDialog1.ShowDialog() != DialogResult.Cancel){
    txtEdit.Font = fontDialog1.Font;
  } // end if
} // end mnuFont
```

The font dialog is displayed in the click event of the mnuFont object. However, this code takes advantage of the fact that the ShowDialog() method returns a value of type DialogResult. (As usual, I knew this only because I looked it up in the online help.) The program simultaneously displays the dialog and examines the result. If the result is not equal to DialogResult.Cancel, the program copies the Font property of the dialog to the Font property of the text box. The logic works like this: There are only two ways to get out of the dialog box—by clicking OK or Cancel. If the user clicks the Cancel button (indicated by a DialogResult.Cancel value), the program should simply move on. If the user chooses anything else (so that the result of the dialog is anything but DialogResult.Cancel),

the program should proceed with the font–changing process.

Trick If you like, you can use this approach of accessing the result of the ShowDialog() method to write your file access code without using the file dialog's methods. I will soon show you this second technique of file dialog access in the section "Storing Entire Objects with Object Serialization."

Using the Color Dialog Control

The code for the color–changing menu items is very similar to the code for changing the font:

```
private void mnuForeground_Click(object sender,
        System.EventArgs e) {
  if (colorDialog1.ShowDialog() != DialogResult.Cancel){
    txtEdit.ForeColor = colorDialog1.Color;
  } // end if
}

private void mnuBackground_Click(object sender,
        System.EventArgs e) {
  if (colorDialog1.ShowDialog() != DialogResult.Cancel){
        txtEdit.BackColor = colorDialog1.Color;
  } // end if
} // end
```

The event handlers are strikingly similar. Both display colorDialog1 and copy the Color property from that dialog to the text box if the user didn't cancel. However, the mnuForeground event copies the color to the text box's ForeColor property, and the mnuBackground event copies the color to the BackColor property of the text box.

Trick If I had insisted on using the event technique with the color dialog, I would have a problem because two different methods need access to the same dialog. Either I would need two color dialogs (which would be wasteful of system resources), or I would have to determine in the event code which method called the dialog (which is possible with the sender parameter, but somewhat clumsy). The approach described in this section seems more appropriate for this particular situation.

Storing Entire Objects with Serialization

Text–based files are very easy to work with as long as the information is not long or complex. However, many applications require manipulation of more complex data. In an object–oriented language such as C#, it makes sense to be able to save and load objects directly, because objects are the primary unit in an OOP program. C# provides a mechanism for easily storing and loading classes—object serialization. In general, you can store any kind of information you can put in a class, and you can easily retrieve the information.

Exploring the Serialization Demo Program

An example will help explain the concept of object serialization. The Serialization Demo program featured in Figure 9.18 illustrates object serialization at work. All the data for a contact is stored in a special class named Contact. The Contact class is stored on the file system and can be loaded from the system. The Save button contains code to send the data to a file, and the Load button retrieves the data from the file.

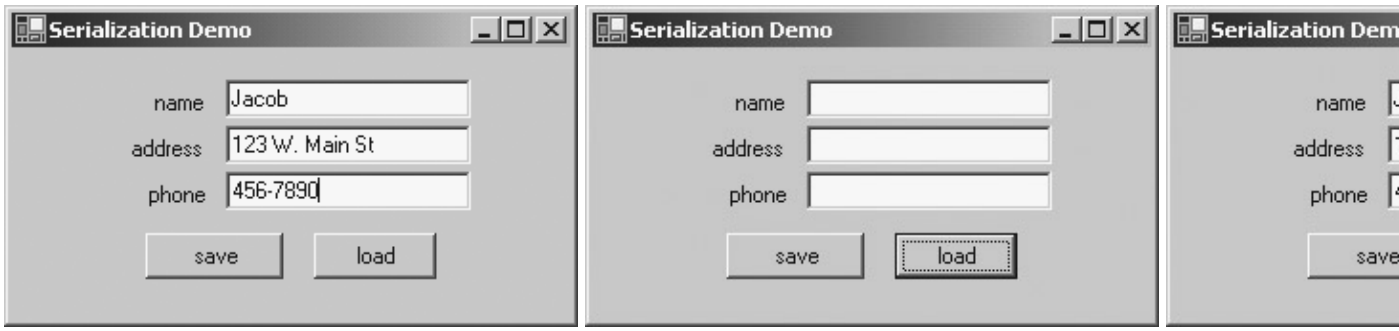


Figure 9.18: The user can enter data into text boxes.

Creating the Contact Class

The Contact class is a very simple class containing three properties. I kept the class simple for demonstration purposes, but any class, even one with methods and events, can be serialized. As you examine the code, you will see only one new element in the class definition:

```
using System;

namespace SerDemo {
  /// <summary>
  /// The Serializable contact class
  /// </summary>

  [Serializable]
  public class Contact {
    private string name = "";
    private string address = "";
    private string phone = "";

    //properties
    public string Name {
      set {
        name = value;
      } // end set
      get {
        return name;
      } // end get
    } // end name prop

    public string Address {
      set {
        address = value;
      } // end set
      get {
        return address;
      } // end get
    } // end address prop

    public string Phone {
      set {
        phone = value;
      } // end set
      get {
        return phone;
      } // end get
    } // end phone prop

    public Contact() {
      //no constructor necessary
    } // end constructor
```

```
  } // end class def
} // end namespace
```

The only new element is the [Serializable] attribute. Attributes are special directives to the compiler. The Serializable attribute indicates that the current class can be serialized to the file system.

Referencing the Serializable Namespace

A class that will load and store serialized objects needs to refer to the IO namespace and a special serialization namespace. I added the following two using statements to the normal array of references at the beginning of SerDemo:

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;
```

System.Runtime.Serialization.Formatters.Binary is a namespace containing classes for storing and retrieving classes.

In the Real World

C# supports *two* types of serialization. The serialization technique described in this chapter is called *binary serialization*. Binary serialization stores data in efficient binary files, but these files are not readable in a text editor and are clumsy to transport across the Internet. The .NET framework purports to support the SOAP (Simple Object Access Protocol), also. This converts an object to an XML format that can be read in a text editor and is easily shared on multiple operating systems. Unfortunately, the SOAP namespace was not included in the preliminary version of C# I used when preparing this book. However, you will find that the techniques for storing a file using SOAP are almost completely identical to the technique described in this chapter for using the binary protocol, even though the underlying technologies are different. If you learn how to do binary serialization, switching to SOAP will be no big deal. Again, encapsulation and polymorphism save the day.

Storing a Class

All the code that manages the storage of the class occurs in the btnStore_Click() method:

```
private void btnSave_Click(object sender, System.EventArgs e) {
  thePerson.Name = txtName.Text;
  thePerson.Address = txtAddress.Text;
  thePerson.Phone = txtPhone.Text;

  FileStream s;
  s = new FileStream("Contacts.bin",
                     FileMode.Create,
                     FileAccess.Write);
  BinaryFormatter bf = new BinaryFormatter();
  bf.Serialize(s, thePerson);
  s.Close();
} // end btnSave_Click
```

The first part of the method copies the parts of the contact from the text boxes to properties of the Contact class. The program then creates a FileStream object. The FileStream is a basic type of stream that can be used for any kind of data, so it is an ideal choice for a serialized class. Because StreamReader is used only to input text and StreamWriter is used only for text input, it was

unnecessary to specify whether they were to be used for input or output. The FileStream can be used for both input and output, so you must specify whether you are opening the class for input or output. FileMode.Create always creates a new file with the specified name (in this case, *Contacts.bin*). In this particular situation, I always want the data I'm writing to overwrite any existing file with the same name. FileAccess.Write indicates that the file should be opened in a way that allows it to be written to. If you open a file with write access, you cannot read from it. Likewise, a file opened for read access cannot be written to. Opening a file with read and write access (FileAccess.ReadWrite) is possible. However, in the simple types of file handling you're learning now, it usually makes sense to open a file for reading only or for writing only and then to close and reopen the file when you want to perform another transaction.

The bf variable is a BinaryFormatter. This is the class that converts the object into a stream of data suitable for file storage. The process of converting an object into a stream of information is called *serialization* because you're converting the data to a form that can be passed to the file a bit at a time. BinaryFormatters have two primary methods: Serialize() and Deserialize(). It won't surprise you that the Serialize() method converts a class to a file–friendly format and the Deserialize() method converts back from the file format to a usable class. The btnSave() method is about saving a class to a file, so I used the Serialize() method. This method takes two parameters: a file stream ready for output and a class to convert. Finally, I close the file stream, and the class data is saved.

Retrieving a Class

The corresponding code to retrieve the class from the file is similar to the btnSave code:

```
private void btnLoad_Click(object sender, System.EventArgs e) {
  FileStream s;
  s = new FileStream("Contacts.bin",
                     FileMode.Open,
                     FileAccess.Read);
  BinaryFormatter bf = new BinaryFormatter();
  thePerson = (Contact)bf.Deserialize(s);
  s.Close();
  txtName.Text = thePerson.Name;
  txtAddress.Text = thePerson.Address;
  txtPhone.Text = thePerson.Phone;
} // end btnLoad_Click
```

To load a class from the disk, the order is reversed. First, I created a file stream. I used the same file name, but this time I used FileMode.Open and FileAccess.Read to indicate that I wanted to open the file for read access. I retrieved the object from the file with the Deserialize() method. This method requires only a stream name as a parameter and returns an object. The type of object returned is unspecified, so I cast it to the Contact class. You have done type casting on primitive variable types, but you might be surprised that the technique works on objects. It does, but only in limited circumstances. In essence, you need to know that the cast will be valid. In this case, I know that I stored a Contact to the file, so I'll have no problem converting the contents of the file to another Contact. I then copied the properties of Contact to the appropriate text boxes.

Returning to the Adventure Kit Program

When I first started thinking about the Adventure Kit project (long before C# was invented), I knew that I wanted a reusable adventure engine. I wanted to have a game engine that was really easy to use with a mouse, and I wanted to encourage users to build their own adventures. The process began by my thinking about how the adventure would be organized. I thought about an adventure

as a series of rooms. Each room could have multiple choices, and the user could wander through the rooms to get a sense of space. A group of rooms would form a dungeon (although the game doesn't have to take place in a dungeon—that just seems like a good collective noun for a bunch of adventure rooms!). The user should be able to move easily between rooms. The game should be edited with an interface similar to the game interface. Because you already saw the game at the beginning of the chapter, you know that I succeeded somewhat. I want to take you through the process, though, because that's where the real thrill of programming is.

In the Real World

If you're an experienced programmer, you might be wondering why I haven't covered random access files yet. The techniques described so far can be used to generate *random access files* (files that can be read in any order, without necessarily going through each element one at a time). You will find that C# provides other tools including XML and ADO data access. These tools provide all the functionality of random access files and are easier to implement. Still, I'll show you a way to store and manipulate several classes at once before this chapter is over.

As usual, most of the code for this chapter's project contains things you have learned in this chapter and earlier chapters. The Adventure Kit is interesting because it has several layers of complexity. The best way to approach this program is to look first at the data structure and then at how that data structure is used in a more complex structure. I'll take you through the game engine itself, which is surprisingly easy to write when you have thought out the data. Next, I'll show you how to build the editor, which is similar to the game engine but requires a little more thought to create. Finally, I'll show you the main program, which attaches the pieces together.

In the Real World

Note that this way of thinking about a program is almost completely opposite of how a user typically sees software. The first thing the user will see is the main screen. The game screen will come next, and for many users, that is all they will ever see. Only the more sophisticated users will try to use the editor, and almost none will think about the underlying data structure. Throughout this book, I've been trying to show you how programmers think, which is very different from how users think. If you want to write interesting programs, you need to practice thinking about your programs from the "data up" instead of the way users usually see things.

Examining the Room Class

The first thing I did when designing the Adventure Kit was turn off the computer. I got out some paper and drew a picture, shown in Figure 9.19. I then thought about how I could describe each room. After a couple iterations, I came up with Table 9.2. If you compare Table 9.2 with Figure 9.19, you will see that Table 9.2 describes the information in Figure 9.19.

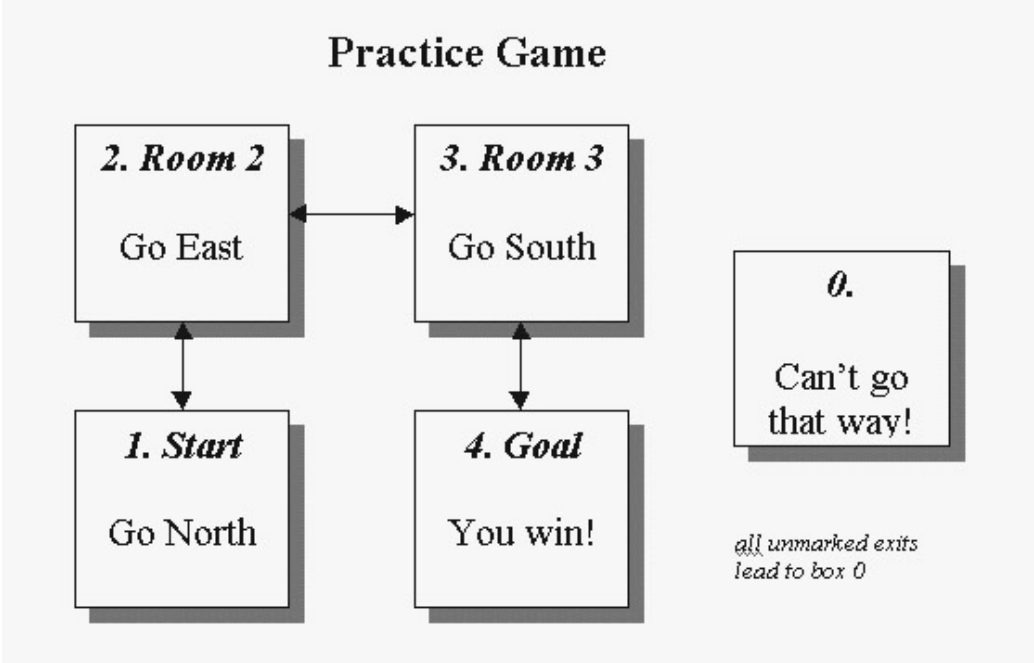


Figure 9.19: I drew a very simple dungeon to get a sense of what kind of data an adventure game requires.

Table 9.2: A Simple Dungeon

Number	Name	Description	N	E	S	W
0	Stuck	Can't go that way	1	0	0	0
1	Start	Go North	2	0	0	0
2	Room 2	Go East	0	3	1	0
3	Room 3	Go South	0	0	4	2
4	Goal	You Win!	3	0	0	0

As you can see from the table, each room has a number, name, and description. In addition, each room has several direction elements. Each direction box indicates which room the user will encounter if he goes in that direction. For example, if the user is in room 2 and goes north or west, he will be sent to room 0, which tells him that he is stuck. If the user goes east from room 2, he will be sent to room 3. If he goes south from room 2, he will end up in room 1. Compare the chart to the diagram in Figure 9.19 to see how the chart describes the diagram.

Trick I'm using the term *room* very loosely here. It's possible that each row of the chart represents an actual room in your game, but that's not necessarily the case. For example, in the Enigma adventure described at the beginning of this chapter, some rooms are actions and some are decisions. Still, having a consistent vocabulary is convenient, so I'll consider each node in the adventure a room.

When you have the chart, you will see a data structure to build. It occurred to me that each row represents a room and that building a room class that encapsulates all the data about a room would be easy.

Hint The table I used to design the Enigma game featured at the beginning of this chapter is included on the CD–ROM as Enigma.doc. Take a look at that document to see the data I used for that somewhat more complex program. Be sure you play the game through first, though, because reading the data will spoil the game for you.

Here's my code for the room class: (I showed only the property code for the name property to preserve space. All the other properties work exactly like the name property, with a very standard get and set procedure. Check out the full code on the CD−ROM for the complete code.)

```
using System;

namespace Adventure
{
  /// <summary>
  /// Basic Data class for Adventure Game.
  /// Dungeon uses an array of these
  /// No methods, just a bunch of properties and a constructor
  /// Andy Harris, 3/11/02
  /// </summary>
  [Serializable]

  public class Room  {

    private string pName;
    private string pDescription;
    private int pNorth;
    private int pEast;
    private int pSouth;
    private int pWest;

    //Properties
    public string Name {
      set {
        pName = value;
      } // end set
      get {
        return pName;
      } // end get
    } // end Name prop

   // Other properties not show.  See CD-ROM for complete code

    //Constructor
    public Room(string name,
                string description,
                int north,
                int east,
                int south,
                int west) {
      Name = name;
      Description = description;
      North = north;
      East = east;
      South = south;
      West = west;
    } // end constructor
  } // end class def
} // end namespace
```

Room is a simple class. It contains six properties, one for each column of the table. The properties surround private instance variables, and the only constructor for the class requires values for every property. I made the Room class serializable because I'm sure that I'll need to save and load it down the road. Now that I have a way for my code to describe the most important part of my program, I've finished the hard part of the overall design.

This strategy of organizing your information into tables and then building a class to represent the table data isn't just for game programming. In fact, it's the key to any kind of programming that involves large amounts of information. Getting a handle on your data is clearly the starting point of writing a good program. If you design your data well, your program will flow towards completion with relative smoothness. If you're sloppy in the way you design data (for example, you don't clearly think through how the user will get from one room to another), you will struggle throughout the entire process. Nothing seems to have a more important effect on a programmer's success than his understanding of the data.

Creating the Dungeon Class

I am proud of my Room class because it will help me with my goal of building a dungeon. However, the Room represents just one row of the chart. I need to represent many rows at once. The easiest way to group the rooms is to build another class that holds an array of rooms. That class is named the Dungeon class:

```
using System;

namespace Adventure
{
  /// <summary>
  /// Class for storing a dungeon. Mainly holds an array of rooms.
  /// Designed to be stored in a serial form.
  /// 3/11/02, Andy Harris
  /// </summary>

  [Serializable]
  public class Dungeon
  {
    private string pName;
    private int pNumRooms = 20;
    private Room[] pRooms;

    public string Name {
      set {
        pName = value;
      } // end set
      get {
        return pName;
      } // end get
    } // end name property

    public int NumRooms {
      //no set - make it read-only
      get {
        return pNumRooms;
      } // end get
    } // end numRooms property

    public Room[] Rooms{
      set {
        pRooms = value;
      } // end set
      get {
        return pRooms;
```

```
      } // end get
    } // end property

    public Dungeon(){
      Rooms = new Room[pNumRooms];
    } // end constructor
  }  // end class def
} // end namespace
```

The Dungeon class has three properties and a constructor. The Name property is the name of the current game. The numRooms property is a read–only property that stores the number of rooms in the current dungeon. I made the numRooms property read–only because it can cause some serious problems if the number of rooms is changed thoughtlessly. I preset the number of rooms at 20, but changing the Dungeon class to handle more rooms would be easy. However, none of the files stored with the 20–room version of the program will work with the new one, and vice versa. It seems that 20 rooms is enough to build complex adventures (such as the Enigma game) without becoming overwhelming.

The most critical property of the Dungeon class is the array of Room objects, named (cleverly enough) Rooms. By having the rooms stored in an array, I gain several important advantages. First, I don't have to worry about adding a room number to the room class, because the index in the array will serve as the room number. Second, accessing each room by its index will be easy. Third, because the array of rooms is part of the Dungeon class, I can store and load all the rooms at once by serializing Dungeon.

The Dungeon class is serializable. Because it includes instances of the Room class, Room must be serializable as well. The combination of Room and Dungeon completes the basic data structure for the game.

Writing the Game Class

It might surprise you that the actual game form is probably the simplest part of the program. All the careful work designing the data makes the game itself quite simple to write.

Creating the Game Form's Visual Design

The Game class is designed around the metaphor of a scroll, with arrows pointing in four directions. Figure 9.20 shows the game form's visual design. The most obvious feature of the game window is the central label named lblDescription. I added a scroll image as the background image of the label and gave it a font that reminded me of a treasure map. The four surrounding labels are used to describe what will happen when the player moves in a particular direction. Each of these labels features a background image as well. I used figures of pointing hands to illustrate the possible positions. Each label also features text that describes the name of the room the user will encounter if he goes in that direction.

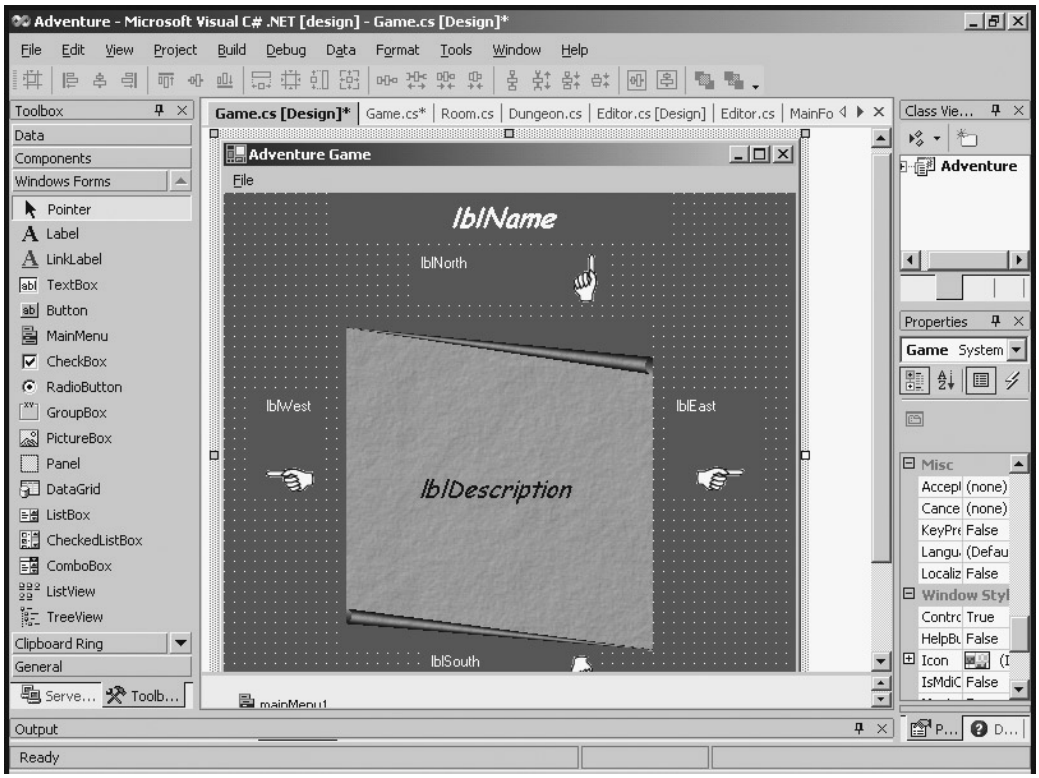


Figure 9.20: The game window features several labels and a menu.

In the Real World

Because this program involves several forms, it is important that they have a unifying visual design. I built a Scroll image in my image editor and placed it on the back of every form, so it looks as though the instructions are written on a treasure map. I also chose similar fonts throughout the program and kept the general layout of the editor and the game screen similar, even though the actual controls are completely different in these two forms. It pays to keep visual unity in your program to reassure the user that he is still using your program even though he changes screens several times.

LblName will hold the name of the current room. The form contains one menu with only two choices. If the user chooses to edit the game, the current form closes, and the editor opens with the current game loaded in it. If the user chooses to quit playing, the program returns to the main screen.

Building the Game Class Instance Variables

The Game class has only two instance variables. Both are used to keep track of the adventure data:

```
private int currentRoom = 1;
public Dungeon theDungeon = new Dungeon();
```

The currentRoom variable is used to specify which room is currently being displayed. theDungeon refers to the current dungeon structure, which, in turn, holds all the room data.

Initializing in Game_Load()

When a Windows program first loads, it automatically calls a Load() method. I decided to add all my initialization code to Game_Load() rather than to the constructor. I like the fact that the constructor

has all the Designer–generated code and my custom initialization goes in the Game_Load() method. The code for Game_Load() simply calls other custom methods:

```
private void Game_Load(object sender, System.EventArgs e) {
  setupRooms();
  showRoom(1);
} // end load
```

The setupRooms() method (as you will see shortly) gives default values to each of the rooms. The showRooms() method takes a room number as a parameter and displays the appropriate room on the form.

Setting Up the Rooms

The setupRooms() method is an interesting method. It's an artifact from the early development process but is still a useful method. It sets up a default game. Mainly, I used setupRooms() before I had the save and load procedures working, to test the basic operation of the program. Later versions of the program made this method unnecessary, but it remains in the code in case I want to use it again:

```
private void setupRooms(){
  //used to set up a 'default' game.
  //Also used to test before editor was finished

  theDungeon.Rooms[0] = new Room(
    "Game Over",
    "You have lost",
    0, 0, 0, 0);

  theDungeon.Rooms[1] = new Room(
    "Start",
    "Go North",
    2, 0, 0, 0);

  theDungeon.Rooms[2] = new Room(
    "Room 2",
    "Go East",
    0, 3, 1, 0);

  theDungeon.Rooms[3] = new Room(
    "Room 3",
    "Go South",
    0, 0, 4, 2);

  theDungeon.Rooms[4] = new Room(
    "You Win!",
    "You have won!",
    3, 0, 0, 0);

} // end setupRooms
```

Trick If you look at the Game class code on the CD–ROM, you will see that it also includes a Main() method. When I started writing this program, I began with the Room and Dungeon classes. Then I wrote the Game class as a standalone program. When I was able to get the basic form of the game working, I was ready to add the editor and main menu classes. It's very common for programs to live through several iterations like this. Even after I added the other forms, I kept in some of the code that allows the Game class to be run as a standalone program, because I might want that functionality again.

Showing a Room

The main way the game communicates with the user is by loading values in the various labels, based on a given room number. The showRoom() method performs this task:

```
private  void showRoom(int roomNum){
  // show a room on the form

  int nextRoom;

  currentRoom = roomNum;
  lblName.Text = theDungeon.Rooms[roomNum].Name;
  lblDescription.Text =
    theDungeon.Rooms[roomNum].Description;

  nextRoom = theDungeon.Rooms[roomNum].North;
  lblNorth.Text = theDungeon.Rooms[nextRoom].Name;

  nextRoom = theDungeon.Rooms[roomNum].East;
  lblEast.Text = theDungeon.Rooms[nextRoom].Name;

  nextRoom = theDungeon.Rooms[roomNum].South;
  lblSouth.Text = theDungeon.Rooms[nextRoom].Name;

  nextRoom = theDungeon.Rooms[roomNum].West;
  lblWest.Text = theDungeon.Rooms[nextRoom].Name;

} // end showRoom
```

The showRoom() method requires a room number as a parameter. It then examines the Rooms array of theDungeon. It extracts the appropriate elements from the current room and copies the values to appropriate parts of the screen. I copied the Name property from the current room to the room name label (lblName). I also copied the Description property over to lblDescription. The Room class stores the indices of the rooms in each direction, but these indices are integers, which mean nothing to the user. Instead, I used the nextRoom variable to determine the index in each direction and then requested the Name property associated with that variable. This results in room names in each direction label.

Responding to Label Events

The user indicates which room he wants to visit next by clicking one of the direction labels. I added code to each of the direction arrows to respond to the user's requests:

```
//label events
private void lblNorth_Click(object sender,
        System.EventArgs e) {
  showRoom(theDungeon.Rooms[currentRoom].North);
}

private void lblEast_Click(object sender,
        System.EventArgs e) {
  showRoom(theDungeon.Rooms[currentRoom].East);
}
private void lblSouth_Click(object sender,
        System.EventArgs e) {
  showRoom(theDungeon.Rooms[currentRoom].South);
}

private void lblWest_Click(object sender,
```

```
        System.EventArgs e) {
  showRoom(theDungeon.Rooms[currentRoom].West);
}
```

The code for all the direction labels follows a common plan. In each case, I simply call the showRoom() method with the index of the correct direction property of the current room.

Creating the Open Game Method

I actually created two distinct versions of the OpenGame() method. The first version was needed when the game program was meant to stand on its own. It calls the fileOpener() to request the file name from the user and then reads a dungeon from the file, using the binary formatter to deserialize the data. It then sets the current room to room number 1 and shows that room. Finally, it closes the file stream.

```
public void OpenGame(){
  //no longer needed
  //read the data from a binary file
  FileStream s;
  BinaryFormatter bf = new BinaryFormatter();
  if (fileOpener.ShowDialog() !=
      DialogResult.Cancel){
    s = new FileStream(fileName, FileMode.Open);
    theDungeon = (Dungeon) bf.Deserialize(s);
    currentRoom = 1;
    showRoom(currentRoom);
    s.Close();
  } // end if
} // end openGame

public void OpenGame(Dungeon passedDungeon){
  theDungeon = passedDungeon;
  currentRoom = 1;
  showRoom(currentRoom);
} // end OpenGame
```

The second version of the OpenGame() method is used when the Game class is run as part of the Adventure Kit. In that case, I decided that the user should choose a game before calling the Game class. As you will see when you examine the MainMenu code, a dungeon will already be loaded in memory when the Game class is started from MainMenu. The new version of OpenGame simply takes a dungeon as a parameter and copies it to theDungeon. It then sets the current room to 1 and shows the room.

> **Trick** Remember, there's nothing wrong with having two versions of the same method, as long as they have different sets of parameters. The OpenGame() method is a good illustration of the power of polymorphism.

Responding to the Menu Events

The game form supports a very simple method. The two menu items allow the user to close the game form and return to the main window or to edit the currently loaded game.

The exit code clears the current form from memory using this.Dispose():

```
private void mnuExit_Click(object sender, System.EventArgs e) {
  this.Dispose();
} // end mnuExit
```

```
private void mnuEdit_Click(object sender, System.EventArgs e) {
  Editor theEditor = new Editor();
  theEditor.Show();
  theEditor.OpenGame(theDungeon);
  this.Dispose();
} // end game_load
```

The Edit menu also closes the game form, but first, it creates an instance of the Editor class, opens the current dungeon in the editor, and displays the Editor class on the screen with its Show() method.

Writing the Editor Class

The adventure program would have been interesting if I had stopped at the Game class. I think that adding the editor makes the game much more interesting, though, because it enables the user community to create many adventures. The Editor class is more challenging than the Game class, but it isn't too tricky.

Creating the Editor Form's Visual Design

Visually, the editor form looks much like the game form but is designed to let the user edit each field. Figure 9.21 shows the editor form's visual layout. The central description for each room is a text box instead of a label, and each direction is represented with a drop–down list box populated with the names of all the rooms in the dungeon. The user navigates through the rooms with Next and Prev buttons. The editor features save and load dialogs, and its menu structure is slightly more complex than the game program because it allows for saving a game, as well as creating a new game from scratch, closing the editor, and playing the current game.

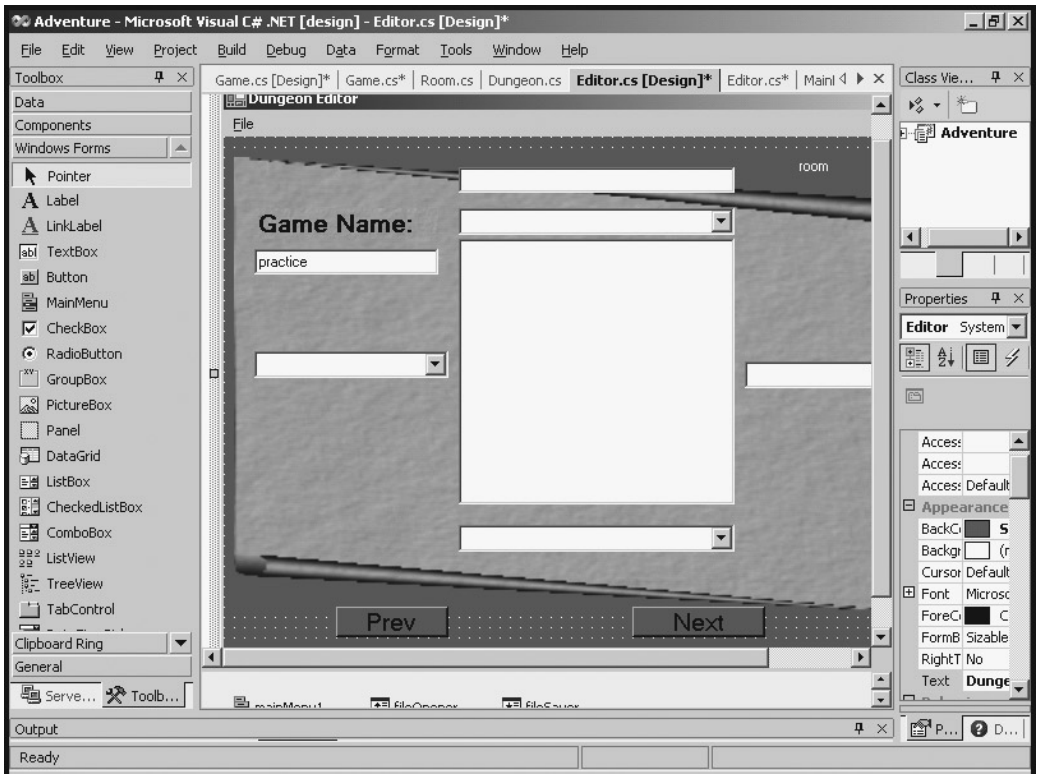


Figure 9.21: The layout for the editor is similar to the game form, but the controls can be edited.

Building the Editor Class Instance Variables

The instance variables for the Editor class are much like those for the Game class. theDungeon is used to store all the game data, and roomNum stores the index of the current room:

```
private Dungeon theDungeon = new Dungeon();
private int roomNum = 1;
```

Initializing in Editor_Load()

As in the game program, I chose to do my own initialization in the Load() event. The setupRooms() method initializes all the rooms to a default value, and setupCombo() assigns the combo boxes the names of the rooms in the dungeon.

```
private void Editor_Load(object sender,
        System.EventArgs e) {
  setupRooms();
  setupCombos();
} // end editorLoad
```

Setting Up the Rooms

The setupRooms() method is used to initialize the rooms. It is called when the class first loads and when the user calls for a new game:

```
private void setupRooms(){
  //initialize rooms
  int i;
  for (i = 0; i < theDungeon.NumRooms; i++){
    theDungeon.Rooms[i] = new Room(
      "room " + Convert.ToString(i),
      "",
      0,0,0,0);
  } // end for loop
} // end setupRooms
```

The method uses a for loop to step through each room in the dungeon and set its values to appropriate default values. I chose to have the room names include a string representation of the room number because I think that it makes editing a game much easier.

Setting Up the Combo Boxes

The user will edit the game by creating a room at a time. In each room, by using a series of combo boxes, the user will determine what happens when the player goes in a particular direction. The combo boxes contain the current list of room names. Each time the user changes a room name, all the combo boxes need to be updated:

```
private void setupCombos(){
  //ensures the combo boxes are up-to-date
  int i;

  //clear the combos
  cboNorth.Items.Clear();
  cboEast.Items.Clear();
  cboSouth.Items.Clear();
  cboWest.Items.Clear();
```

```
  //repopulate the combos
  for (i = 0; i < theDungeon.NumRooms; i++){
    cboNorth.Items.Add(theDungeon.Rooms[i].Name);
    cboEast.Items.Add(theDungeon.Rooms[i].Name);
    cboSouth.Items.Add(theDungeon.Rooms[i].Name);
    cboWest.Items.Add(theDungeon.Rooms[i].Name);
  } // end for loop

  //preselect room zero
  cboNorth.SelectedIndex = 0;
  cboEast.SelectedIndex = 0;
  cboSouth.SelectedIndex = 0;
  cboWest.SelectedIndex = 0;

} //end setupCombos
```

The easiest way to update the combos is to clear them out completely and rebuild them. The Items property of the combo box has a Clear() method, which performs this task admirably. The method then steps through each room, adding each room's name to each combo box. Finally, the method presets each combo so that it points to room 0.

Trick As I developed examples for this chapter, I started evolving my own convention about the games developed with this kit. I reserved room 0 as the "You can't go there" room and used room 1 as the basic startup room. For that reason, when you call up an adventure in the editor, it will begin in room 0, but if you load the same file into the Game interface, it will begin in room 1.

Showing a Room

The user will be able to move between the rooms with the command buttons at the bottom of the screen. It is important to be able to display any given room:

```
private void showRoom(){
      //displays a room in editor

  setupCombos();
  txtName.Text = theDungeon.Rooms[roomNum].Name;
  txtDescription.Text =
      theDungeon.Rooms[roomNum].Description;

  cboNorth.SelectedIndex = theDungeon.Rooms[roomNum].North;
  cboEast.SelectedIndex = theDungeon.Rooms[roomNum].East;
  cboSouth.SelectedIndex = theDungeon.Rooms[roomNum].South;
  cboWest.SelectedIndex = theDungeon.Rooms[roomNum].West;
  lblRoomNum.Text = "room " + Convert.ToString(roomNum);
} // end showRoom
```

The first task is to reset the combo boxes to take into account any changes in the room data. After that, the method copies the name and description to the appropriate text boxes. Rather than copy the direction values into the database, these numeric values are used to set the index of the combo boxes to the appropriate value, which will display the room number associated with the room. For example, if the North value of the current room is 3, the third element of the combo box will be the name of room 3, because of the setupCombos() call. Setting cboNorth.SelectedIndex to 3 causes the third element of the combo box to appear, which will be the name of room 3.

Storing a Room

Storing a room is the logical opposite of saving the room. Basically, the method copies values from the form elements to the current room. Note that the selected index of the direction combos is set, not the text value:

```
private void storeRoom(){
  //stores the current room to the database
  theDungeon.Rooms[roomNum].Name = txtName.Text;
  theDungeon.Rooms[roomNum].Description =
             txtDescription.Text;

  theDungeon.Rooms[roomNum].North = cboNorth.SelectedIndex;
  theDungeon.Rooms[roomNum].East = cboEast.SelectedIndex;
  theDungeon.Rooms[roomNum].South = cboSouth.SelectedIndex;
  theDungeon.Rooms[roomNum].West = cboWest.SelectedIndex;
} // end storeRoom
```

Trick The relationship between the directional values and the list boxes illustrates an important point. The numeric values are convenient from the programmer's perspective because they are unambiguous. It is very easy to see which room number to display next if the user clicks the North label. However, human users much prefer text or visual cues to numeric values. The value of the combo boxes is how the way to bridge this gap. When I'm interested in the actual numeric value associated with a direction, I use the selectedIndex property. The user can just deal with the string values without knowing or caring that the *position* of something in the list box is what matters to the program, not what it *says*.

Responding to the Next and Prev Button Events

The Next and Prev buttons are used to let the user navigate between records in the game editor. They do a lot of work, but most of that work is encapsulated into the storeRoom() and showRoom() methods you've already seen:

```
private void btnPrev_Click(object sender,
        System.EventArgs e) {
  storeRoom();
  roomNum--;
  if (roomNum < 0){
    roomNum = 0;
  } // end if
  showRoom();
} // end btn prev

private void btnNext_Click(object sender,
        System.EventArgs e) {
  storeRoom();
  roomNum++;
  if (roomNum >= theDungeon.NumRooms){
    roomNum = theDungeon.NumRooms - 1;
  } // end if
  showRoom();
} // end btnNext click
```

The btnPrev_Click event stores the current room to preserve any changes that have happened. It then decrements the room number and checks whether the room number is less than 0. If so, the room number is set to 0. The call to showRoom() shows the room, based on the current room number.

The btnNext_Click event works in very much the same way, except that it increments the variable, rather than decrements it, and checks whether the value is larger than or equal to the number of rooms in the dungeon. If so, roomNum is set to the number of rooms—1. (Remember, arrays begin with an index of 0, so the largest possible value will be theDungeon.NumRooms – 1.

Saving the Adventure

The adventure game is saved with the now familiar binary serialization technique. The method starts by pulling the game's name from its textbox and assigning the result to the Name property of theDungeon. Then the current room is stored in case changes have been made but the Next or Prev button hasn't been clicked. The data is stored in the file, using a FileStream and a BinaryFormatter:

```
private void mnuSaveAs_Click(object sender,
        System.EventArgs e) {
  //get the game's name
  theDungeon.Name = txtGameName.Text;
  //store the current room
  storeRoom();
  //write the data out to a binary file
  FileStream s;
  BinaryFormatter bf = new BinaryFormatter();
  if (fileSaver.ShowDialog() != DialogResult.Cancel){
    s = new FileStream(fileSaver.FileName, FileMode.Create);
    bf.Serialize(s, theDungeon);
    s.Close();
  } // end if
} // end mnuSave
```

Loading the Adventure

Loading the adventure works just as it did in the Game class, using binary serialization. After I loaded the game in memory, I set the room number to 1 and showed the room:

```
private void mnuOpen_Click(object sender,
        System.EventArgs e) {
  //read the data from a binary file
  FileStream s;
  BinaryFormatter bf = new BinaryFormatter();
  if (fileOpener.ShowDialog() != DialogResult.Cancel){
    s = new FileStream(fileOpener.FileName, FileMode.Open);
    theDungeon = (Dungeon) bf.Deserialize(s);
    roomNum = 0;
    showRoom();
    s.Close();
  } // end if
} // end mnuOpen
```

Opening a Game

As in the Game class, it will be possible for the MainForm to start the editor remotely, so I added an OpenGame() method that will start the editor when given a Dungeon as a parameter:

```
public void OpenGame(Dungeon passedDungeon){
  theDungeon = passedDungeon;
  roomNum = 0;
  txtGameName.Text = theDungeon.Name;
  showRoom();
```

```
} // end Open_game
```

The openGame() method simply copies the passed dungeon parameter to theDungeon, initializes the room number to 0, copies the dungeon name to the appropriate text box, and shows the room.

Exiting, Playing, and Creating a New Game

The other menu event handlers are (as usual) standard fare. When the user chooses to exit the editor, I use this.Dispose() to eliminate the form from memory.

If the user wants to create a new game, the setupRooms() method does most of the work, but I also reset roomNum to 0 and showed room 0. Finally, I reset the txtGameName text box to empty.

The mnuPlay_Click() method directly calls the game screen so that the user can immediately play whatever adventure he has been working on without having to return to the main form first. It opens a new instance of the Game class, shows the form, and opens up the current dungeon with a call to theGame.OpenGame(). Finally, the method disposes the current (Editor) class to get it out of the way.

```
private void mnuExit_Click(object sender,
  System.EventArgs e) {
  this.Dispose();
} // end mnuExit

private void mnuNewGame_Click(object sender,
        System.EventArgs e) {
  setupRooms();
  roomNum = 0;
  showRoom();
  txtGameName.Text = "";
} // end mnuNewGame

private void mnuPlay_Click(object sender,
        System.EventArgs e) {
  Game theGame = new Game();
  theGame.Show();
  theGame.OpenGame(theDungeon);
  this.Dispose();
} // end mnuPlay
```

Writing the MainForm Class

The MainForm class is the user's initial entry and exit point to the program. It serves mainly as a control center, routing the user between the other forms. Although it appears to the user to be the main part of the program, it is actually the simplest part of the project, and the last part I designed. Each button calls up the appropriate form or closes the program altogether.

Creating the Main Form's Visual Design

The primary purpose of the main form is to tie the rest of the program together. Ironically, the form is most effective if the user spends very little time on it at all. For this reason, all the main actions belong to prominent command buttons that dominate the form. To keep the program appearance unified, I assigned the same scroll background to each button and to a picture box on the form. The form also has a label indicating which game, if any, is currently loaded in memory.

Building the MainForm Class Instance Variables

The MainForm class uses instance variables to control each form, the adventure data, and a file name for the current game:

```
//classes for game and editor screens
private Editor theEditor = new Editor();
private Game theGame = new Game();
private Dungeon theDungeon = new Dungeon();
private string gameFile = "";
```

Loading the Dungeon

The MainForm class is capable of loading a dungeon class before calling the other classes. This ensures that the Game class is never called without a valid dungeon in memory. Also, this makes debugging your adventures easier because you can simultaneously call a game window and an editor window to see how your game plays while you examine it in the editor.

```
private void btnLoad_Click(object sender,
        System.EventArgs e) {

  //read the data from a binary file
  FileStream s;
  BinaryFormatter bf = new BinaryFormatter();
  if (fileOpener.ShowDialog() != DialogResult.Cancel){
    gameFile = fileOpener.FileName;
    s = new FileStream(fileOpener.FileName, FileMode.Open);
    theDungeon = (Dungeon) bf.Deserialize(s);
    s.Close();
    lblCurrent.Text = "Game: " + theDungeon.Name;
  } // end if
} // end btnLoad
```

The binary serialization technique, which has been such a workhorse in this program, is called into service one more time. As usual, the value of the file is copied to a Dungeon instance after being deserialized.

This theDungeon variable will be used to call the OpenGame() methods of the other two forms.

Playing the Game

If the user wants to play a game, the program first checks whether a game has been loaded. If not, it asks the player to do so. If a game has been loaded, starting up the game window is a simple process:

```
private void btnPlay_Click(object sender,
        System.EventArgs e) {
  if (gameFile != ""){
    theGame = new Game();
    theGame.Show();
    theGame.OpenGame(theDungeon);
  } else {
    MessageBox.Show("Please load a game first");
  } // end if
} // end btnPlay
```

If a game has been successfully loaded into the program, the value of the gameFile variable will be

something besides its starting value of "". In that case, the program creates an instance of the Game class, shows the form, and opens the game stored in the current dungeon. If the value of gameFile is still null, the method reminds the user to load a game before trying to play it.

Editing the Game

The process for opening the editor is simpler than for opening the game because it's reasonable for the user to open up the editor without a game in memory, especially if the user wants to create a new game from scratch.

```
private void btnEdit_Click(object sender,
        System.EventArgs e) {
  theEditor = new Editor();
  theEditor.Show();
  if (gameFile != ""){
    theEditor.OpenGame(theDungeon);
  } // end if
} // end btnEdit
```

The method simply creates a new instance of the Editor class, displays the form, and opens up the game if it is already in memory.

Summary

Your programs are beginning to take on more challenging and interesting dimensions. You have added some very useful tools to your bag of tricks in this chapter. You learned how to store and load plain text files. You learned how to use file streams, stream readers, and stream writers to manipulate files. You added menu structures—complete with hotkeys, shortcuts, and multiple levels—to your programs. You learned how the standard dialogs can query the user for fonts, colors, and file names. You practiced binary object serialization to store entire objects to files. You put all these things together in an interesting multiform application.

Challenges

- Add a Save feature to the game editor. Save usually works just like Save As if the file has not yet been saved. However, if the program recognizes a file name for the current file, it automatically saves that file without calling up a dialog box.
- Add menus to the Visual Critter program from Chapter 6, "Creating a Windows Program."
- Modify the Mini Adventure program (the game where you added words to a text file) from Chapter 1 "Basic Input and Output: A Mini Adventure," so that it can read a text file from a disk. Allow the user to put in special tags (such as <noun>). When the program encounters a tag, have it ask for a noun and place it in a variable. Perhaps add an editor so that the player can save and load multiple stories.
- Add object serialization to the Snowball Fight program from Chapter 5, "Constructors, Inheritance, and Polymorphism." Allow the user to modify various characteristics of a robot–controlled fighter. Then store and load these objects to the drive system.
- Extend the Serialization Demo program in this chapter to be a small database. Extend the Contact class so that it has the properties you want, and build a class like Dungeon to store a list of contacts.

Chapter 10: Chapter Basic XML: The Quiz Maker

Since the late 1990s, there has been a great deal of interest in a technology called *XML (eXtended Markup Language)*. XML has promised to transform many types of programming, and support of XML is one selling point of the .NET framework. You will now learn how to incorporate XML in your programs. In this chapter you will learn how to do the following tasks:

- Recognize the basic rules of XML syntax.
- Associate various parts of an XML document with the appropriate .NET classes.
- Navigate through the elements and nodes of an XML document.
- Store data in an XML structure.
- Retrieve data from an XML structure.
- Store and load XML documents to files.

Introducing the Quiz Maker Game

I wrote a simple quiz game to illustrate the use of XML. The quiz program opens with a main screen that calls two other forms (see Figure 10.1). The quiz form allows the player to take a quiz, and the editor form allows the user to create and modify quizzes. However, rather than rely on object serialization as a storage technique as the Adventure Kit game does, the quiz files are stored as XML documents. XML provides several advantages, which are described in the upcoming section "Investigating XML."

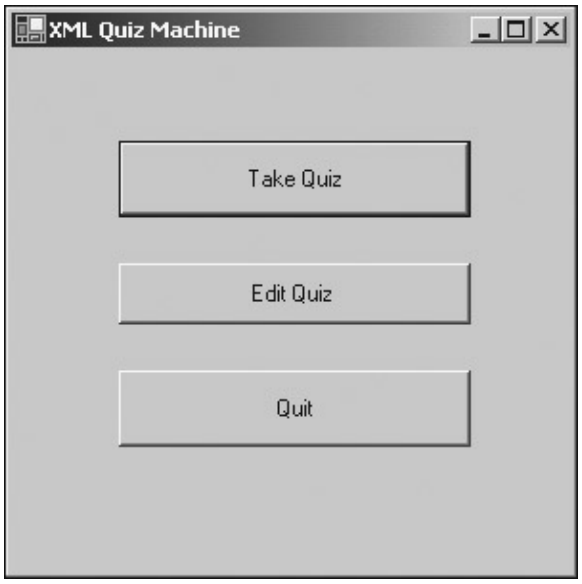


Figure 10.1: The main screen allows quick access to the other forms of the quiz program.

Taking a Quiz

The quiz program serves a series of multiple–choice questions. The user can select an answer with radio buttons and can move between questions with the Next and Prev buttons. Figure 10.2 shows the quiz program in action.

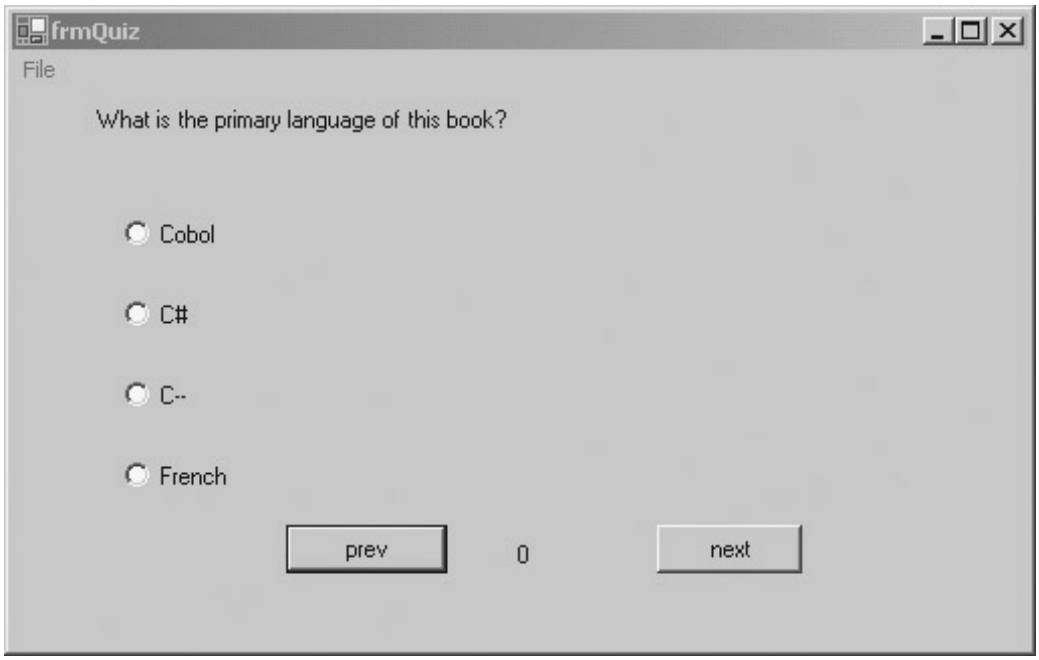


Figure 10.2: The quiz interface is quite simple.

Creating and Editing Quizzes

As in the adventure game from Chapter 9, “File Handling: The Adventure Kit,” the user can edit existing quizzes and create new ones. Figure 10.3 shows the quiz editor. On the surface, the Quiz game is much like the Adventure Kit game. However, the Quiz game uses an entirely different technique for handling the underlying information. I deliberately made these two programs similar in other features so that you can concentrate on the underlying technical differences.

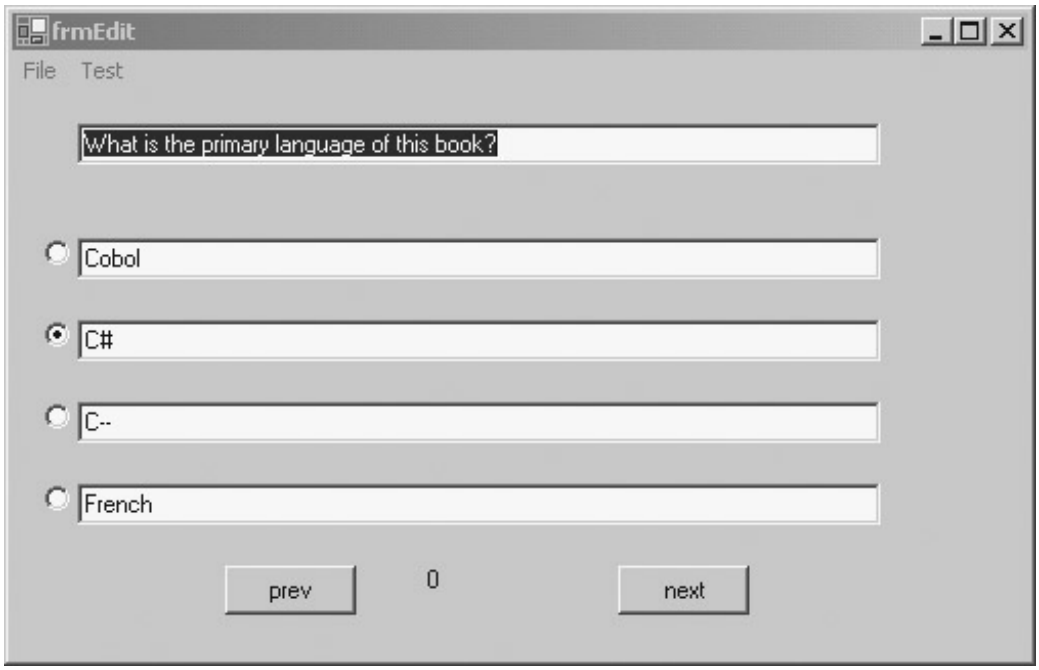


Figure 10.3: The user can enter quiz data using an editor, much like the Adventure Kit game.

Investigating XML

The Quiz game stores and manages quiz data using a format called *eXtended Markup Language (XML)*. XML has become an important technology in the past few years. XML is often used to share

information between programs. You will likely come across XML data somewhere in your programming travels, so you should know how to work with it. Fortunately, C# and .NET provide powerful tools for working with XML data.

Defining XML

XML can be a bewildering topic because it is a wide−reaching standard and implementation tools abound. However, the basic concepts behind XML are not as complex as they might seem.

XML is a technology for describing data. An *XML document* is a file that contains data. In addition to the data itself, an XML document contains information about the data (often called *metadata*). An XML document is written in plain text that can easily be distributed over the Internet. It uses a series of tags to describe the various data elements, much as HTML uses tags to describe the parts of a Web page. The interesting thing about XML is its capability to generate new tags to describe any kind of information.

A *tag* is simply a word inside angle brackets (< >). Most elements in XML have both a beginning tag and an ending tag. A tag can contain text (or other types of data) or other tags.

XML is closely related to HTML, the language of Web pages. XML and HTML share a common ancestor, SGML (Standard Generalized Markup Language), and they demonstrate a strong family resemblance. If you write any HTML code or look at the code behind a Web page, you will find XML to be very similar in its general design. HTML is designed for only one specific job—to format Web pages. XML is designed to let you describe any kind of data you want.

In the Real World

In this chapter I am focusing on how you can use and define your own custom version of XML, but interesting and powerful predefined XML languages are available. Most of these languages are designed to simplify working with a particular type of information or data. Some interesting XML languages are

- **SMIL (Synchronized Multimedia Integration Language).** This language defines multimedia presentations. It is used to generate slide shows, synchronize captions with streaming audio and video content, and manage other kinds of multimedia information.
- **SVG (Scalable Vector Graphics).** This language allows authors to create vector−based graphics, which are often much more compact and flexible than traditional graphics schemes. It is supported by an international standards body, the W3C (World Wide Web Consortium), but not by Microsoft.
- **VML (Vector Markup Language).** This language is Microsoft's answer to SVG. It is another powerful, vector−based system for creating graphics in documents and Web pages.
- **CML (Chemical Markup Language).** This language is dedicated to simplifying the drawing of chemical figures, which has long been a challenge for those who frequently write about chemistry.
- **SOAP (Simple Object Access Protocol).** This language enables objects to communicate across the Internet.

An example will help make sense of all this theory. The quiz program you saw at the beginning of the chapter is based on a form of XML I invented just for this chapter. (That's the fun part of XML—you get to invent new languages.) My new language will describe a quiz. Take a look at the XML code that describes the sample test:

```
<?xml version="1.0" encoding="utf-8"?>

<test>
  <problem>
    <question>What is the primary language of this book?</question>
    <answerA>Cobol</answerA>
    <answerB>C#</answerB>
    <answerC>C--</answerC>
    <answerD>French</answerD>
    <correct>B</correct>
  </problem>

  <problem>
    <question>What does XML stand for?</question>
    <answerA>eXtremely Muddy Language</answerA>
    <answerB>Xerxes, the Magnificent Chameleon</answerB>
    <answerC>eXtensible Markup Language</answerC>
    <answerD>eXecutes with Multiple Limitations</answerD>
    <correct>C</correct>
  </problem>

  <problem>
    <question>Which command sends a message to the user?</question>
    <answerA>Messagebox.Show()</answerA>
    <answerB>sendMessage()</answerB>
    <answerC>Alert()</answerC>
    <answerD>HeyUser()</answerD>
    <correct>A</correct>
  </problem>
</test>
```

By examining the quiz code, you can see that it defines a structure for the quiz. The first line of the code describes the file as XML code and defines the encoding type:

```
<?xml version="1.0" encoding="utf-8"?>
```

Almost every XML file you see begins with a similar line. The utf−8 encoding refers to the Unicode text−formatting scheme used by C# and most other modern languages to support international languages. The line begins with <? and ends with ?> to indicate that this is a special header line. It is required in most XML documents.

Trick The terms version and encoding are *attributes* of the XML tag. Each tag can have attributes that modify the data in some way. For example, the <img> tag in HTML has attributes to modify the width and height of the image. If you design an XML document, your tags can have attributes. Working with attributes is easy, but to keep things simple in this introductory chapter, I decided to use them only where they are mandated, as in the XML tag in the preceding code.

If you examine the quiz's structure, you will quickly see some patterns emerge. The document is composed of nested structures. The bulk of the document is encased inside a <test> </test> pair. Inside that are a series of <problem> </problem> sets. Each problem consists of a <question>, several <answers>, and a <correct> element. You might also see the structure as a hierarchy, as illustrated in Figure 10.4.

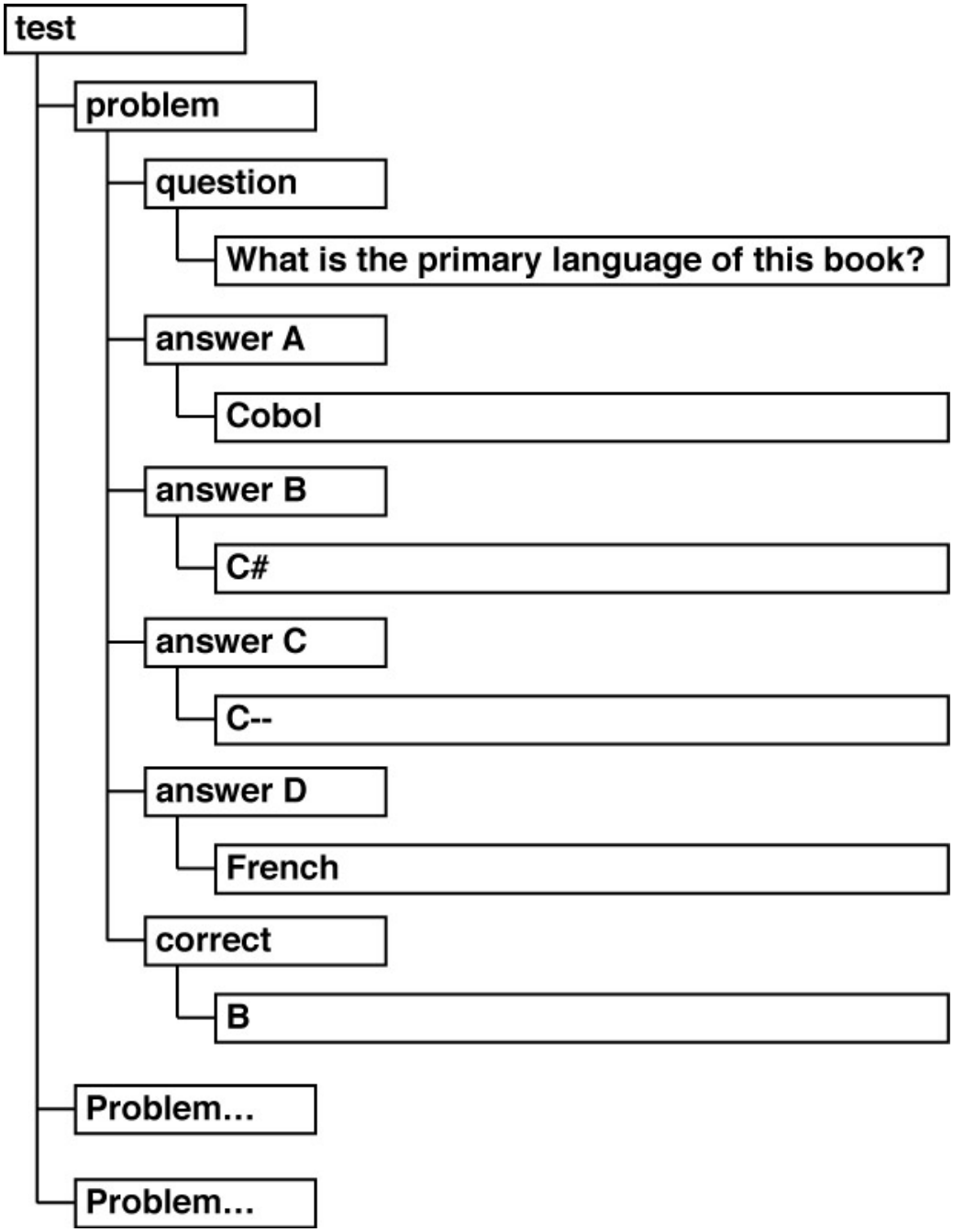


Figure 10.4: The test contains problems, which can contain a question, answers, and the correct answer.

It's important to note that the tags describe what *kind* of data is represented, rather than how data is to be presented on the screen. An important aspect of XML is how it focuses on the meaning of the data alone. XML data is intended to be used by many programs written in different languages on different platforms, so it is up to the program that uses the data to determine how exactly the data will be displayed.

In the Real World

The original intent of HTML was to describe only the meanings of various text elements in the context of a Web page rather than to determine how a page is to be depicted on the monitor. (In fact, a tag such as <i> is considered rude by HTML purists, who prefer the <emphasis> tag because it describes what kind of text is being encased rather than how to display it.) HTML documents are

meant to be displayed on many kinds of devices, including cell phones and PDAs, which cannot always handle all the complex formatting commonly indicated on modern Web pages. For this reason, it is still smarter to use HTML to determine what the text *means* rather than how it is *displayed*. It is considered more appropriate to use style sheets to determine exactly how the data is to be displayed. XML documents can be displayed using style sheets, also, but programmers can do even more. When you learn how to read and manipulate an XML document in this chapter, you will be able to display XML data however you want.

Learning XML Syntax Rules

You must follow a set of rules when creating an XML language. In this sense, XML is stricter than HTML. However, these rules are very easy to learn and allow tremendous flexibility.

All Tags Are Lowercase

In HTML, you can use or <CENTER>. In an XML language, you are required to use lowercase letters for any tag you define. However, to separate words, you can use mixed case, such as <myTag> or <questionNumber>.

All Tags Have an Ending Tag

If you define a tag, it must have a corresponding ending tag. The ending tag is just like the start tag, except for a slash (/) inside the angle brackets. For example, you can define a <note> tag, but you must also have a </note> tag. Tags that do not usually have an ending tag (such as the <img> tag in HTML, which defines an image) can have a slash at the end of the tag. For example,

```
<img src = "myImage.gif"/>
```

is legal in XML, but

```
<img src = "myImage.gif">
```

is not legal.

All Attribute Values Are Enclosed in Quotes

A tag can include an attribute to modify the tag's meaning. For example, in the test data, you can have multiple–choice questions, short–answer questions, and true–or–false questions. You can extend the problem tag to indicate the type of question. For example, you could write

```
<problem type = "mc">
```

to indicate a multiple–choice question

```
<problem type = "tf">
```

or to indicate a true–or–false question.

In this case, type is an attribute, and the value of the type attribute could be mc or tf.

Creating an XML Document in .NET

You can create any kind of XML document you want in order to describe any kind of data you want to work with. However, XML works best with hierarchical types of data structures that can be organized in an outline form. The first step of defining an XML language is to look at the data you are working with and try to organize it into an outline form. For the quiz program, I realized that a quiz is made up of several problems. (I avoided the use of the term *question* to describe the major element of a quiz because I wanted to use *question* to describe the question being asked.)

Each problem consists of a question, four possible answers, and the correct answer. To simplify the example, I decided to work only with multiple–choice questions with four answers. When I had decided what kinds of tags I would have, I started to build a sample document. You can use any text editor you want to build an XML document, but Visual Studio comes with a very nice XML document editor that makes the task simple. To use the XML editor, open up a project in the editor, and choose Add New Item from the Project menu. Select XML Document from the resulting dialog to open the XML editor. If you already have an XML document that you'd like to open in the editor, choose Open File from the File menu, and select the XML file from the drive system. (Alternatively, you can choose Add Existing Item from the Project menu. Opening a file doesn't necessarily add the file to your project, but adding an existing item does.) Figure 10.5 shows an XML file being written in the Visual Studio XML editor.

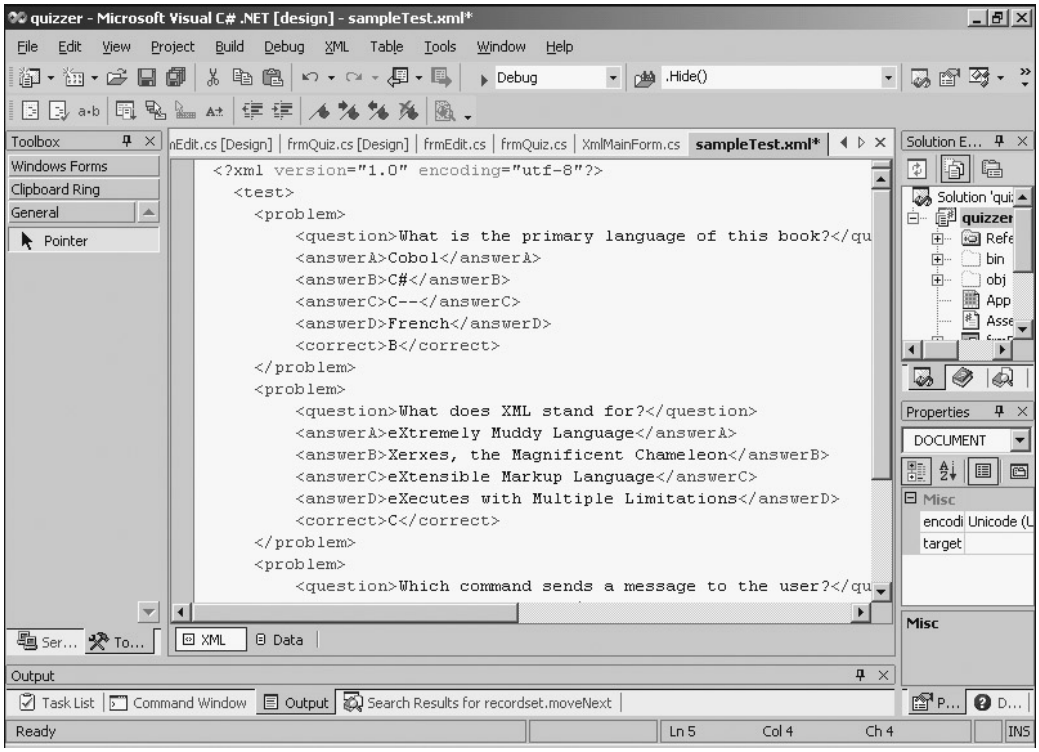
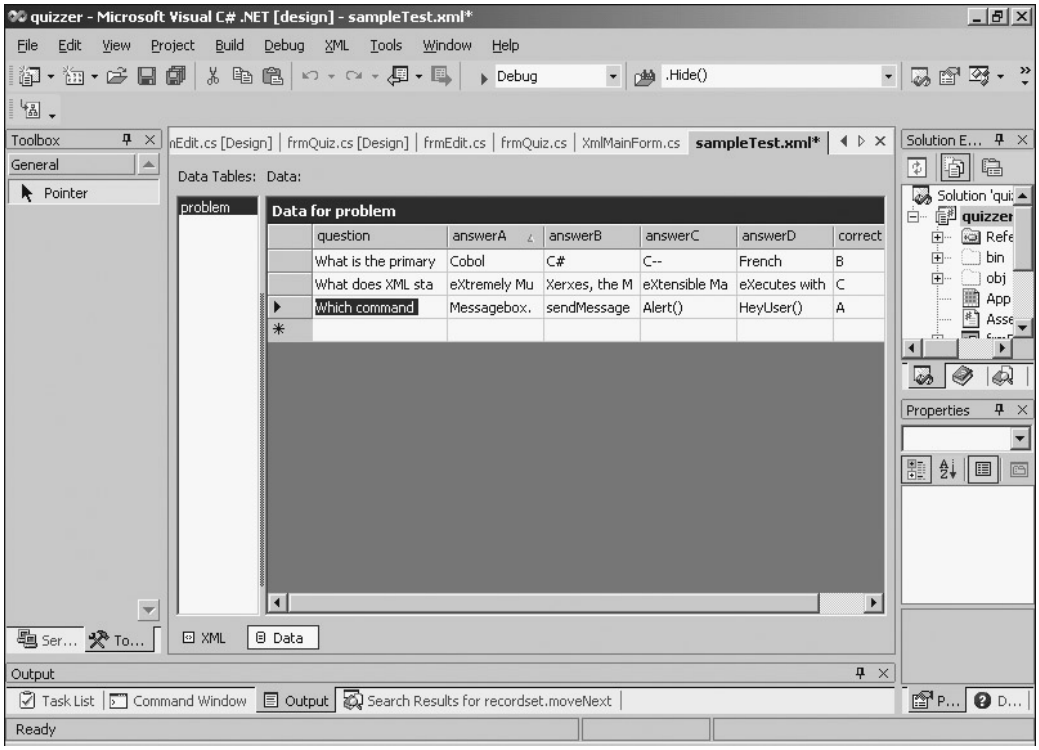


Figure 10.5: The Visual Studio XML editor automatically indents your code and creates a closing tag for each opening tag.

The XML editor included with Visual Studio makes writing XML easy because it automatically indents your XML code and, each time you create a tag, it creates an ending tag. The editor also color–codes the XML much like normal C# code, which helps you to separate the XML code from the actual data. Visual Studio also includes a very handy editor for writing your own XML schema, but you will not need it for this brief introduction to XML. After you create the basic framework of your data structure, you can click the data tag at the bottom of the XML code window to switch to a table view (see Figure 10.6).


quizzer - Microsoft Visual C# .NET [design] - sampleTest.xml*
File Edit View Project Build Debug XML Tools Window Help
Debug
.Hide()
Toolbox
General
Pointer
nEdit.cs [Design] | frmQuiz.cs [Design] | frmEdit.cs | frmQuiz.cs | XmlMainForm.cs | sampleTest.xml*
Data Tables: Data:
problem
Data for problem
question answerA answerB answerC answerD correct
What is the primary Cobol C# C-- French B
What does XML sta eXtremely Mu Xerxes, the M eXtensible Ma eXecutes with C
Which command Messagebox. sendMessage Alert() HeyUser() A
XML Data
Solution E...
Properties
Output
Task List Command Window Output Search Results for recordset.moveNext
Ready
Figure 10.6: After you define your data set, you can enter it in, just like a database.
Visual Studio can automatically convert your XML structure into a table for easy data entry. It allows you to enter your data quickly and accurately without having to repeat all the XML tags. This gives you the flexibility of XML with the easy data access of a more formal database. In Chapter 11, “Databases and ADO.NET: The Spymaster Database,” you will learn more about the relationship between XML and databases in .NET.

Creating an XML Schema for Your Language

If you’re going to reuse your language or you expect other people to use it, you should have a formal definition of the rules of your language. There has been some debate among the XML community about how this should be done, but .NET provides a solution that is extremely simple and elegant. When you load or create XML code in the IDE XML editor, a new XML menu appears on the menu bar. After you define your first set of data, you can choose Schema from the XML menu visible in the XML editor. This creates a new XML document that describes how your data works. The following XML code illustrates the schema automatically generated for the quiz document:

```
<?xml version="1.0" ?>
<xs:schema id="test"
  targetNamespace="http://tempuri.org/sampleTest.xsd"
  xmlns:mstns="http://tempuri.org/sampleTest.xsd"
  xmlns="http://tempuri.org/sampleTest.xsd"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
  attributeFormDefault="qualified"
  elementFormDefault="qualified">
<xs:element name="test"
  msdata:IsDataSet="true"
  msdata:EnforceConstraints="False">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="problem">
          <xs:complexType>
```

```
            <xs:sequence>
              <xs:element name="question" type="xs:string" minOccurs="0" />
              <xs:element name="answerA" type="xs:string" minOccurs="0" />
              <xs:element name="answerB" type="xs:string" minOccurs="0" />
              <xs:element name="answerC" type="xs:string" minOccurs="0" />
              <xs:element name="answerD" type="xs:string" minOccurs="0" />
              <xs:element name="correct" type="xs:string" minOccurs="0" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The meaning of this code is beyond the scope of an introduction to XML, but the code is automatically generated, so you can use it even if you don't know exactly what it's doing.

You can also choose Validate from the XML menu to ensure that your data follows the guidelines generated by the schema. For a beginner, creating a schema and validating your documents are not necessary because your first attempts at XML code will probably be simple and nobody but you will use your particular XML dialect. If you write an XML language that others will use, you will want to explore data validation because it can prevent many kinds of data errors. In this chapter most of the XML will be generated automatically by the programs, so there is no need to validate it.

Investigating the .NET View of XML

The .NET framework defines a set of classes that map to an XML document and its constituent parts. An XML document is essentially seen as a tree. The document itself is the base of the tree. The document contains a series of *nodes*. .NET has a class to define the Node element. Each pair of tags (such as <question> and </question>) is considered an *element*. The information between the tags is the *data* (usually text). The .NET environment provides three classes that are critical for using XML:

- The XmlNode class defines the basic characteristics of any node in an XML document.
- The XmlElement class extends XmlNode and adds a few methods for dealing with attributes.
- The XmlDocument class also extends XmlNode, but it adds several methods, mainly for storing and loading documents and creating new nodes.

Essentially, you work with an XML document in .NET by defining an XmlDocument, which is mapped to a specific document on the drive system. All the tags in the document are instances of the XmlElement class. Because both XmlDocument and XmlElement are derived from XmlNode, they also share the characteristics of the XmlNode class.

Exploring the XmlNode Class

The XmlNode class describes each node in a document. A node can be an entire XML document, an element (a pair of tags), or data (the text inside a pair of tags). The XmlNode class has properties and methods that enable you to figure out what type of node you are working with. It also features the capability to extract and modify the data in a node. Table 10.1 presents important properties and methods of the XmlNode class.

Table 10.1: Selected Members of the XmlNode Class

Element	Type of Element	Description	Example
ChildNodes	Property	A collection of all the children of the node	MessageBox.Show (theNode. ChildNodes[2].InnerText); //shows the text of child node 3 of the current node
FirstChild	Property	The first child of a node	MessageBox.Show (theNode. FirstChild.InnerText); //shows the text of the first child of the current node
InnerText	Property	Gets or sets the text of this node and all its children	MessageBox.Show (theNode.InnerText); //shows the text value of the current node
Name	Property	Returns or sets the name of the node	MessageBox.Show (theNode. Name); //displays the name of the current node
NextSibling	Property	Returns the next node at the current level of the hierarchy (or null)	theNode = theNode. NextSibling; //moves the node to its next sibling or sets the node to null
ParentNode	Property	Returns the parent of the current node	theNode = theNode.ParentNode; //sets the node to its parent
AppendChild(node)	Method	Adds a child node to the end of this node's children	theNode.AppendNode(newNode); //adds newNode to theNode's children
Clone()	Method	Creates a copy of this node	theNode.ParentNode. AppendNode(theNode.Clone()); //adds a copy of this node at the same level as the node
RemoveChild(node)	Method	Removes a child node from the current node	theNode.Remove (theNode. FirstChild); //removes the first child from the node

The XmlNode class is rarely used directly, but it has two descendants that form the foundation of all XML documents. The XmlDocument class describes an entire document, and the XmlElement class describes an *element*, which is the part of an XML document surrounded by a pair of tags. Much of the functionality of the XmlDocument and XmlElement classes is inherited from the XmlNode class, but the XmlDocument class has a few important properties and methods of its own, which are particular to a document.

Exploring the XmlDocument Class

The XmlDocument class describes an XML document. It features important properties and methods for working with the entire document. It has save and load methods, which allow you to save and load a document directly, without needing to use streams. The XmlDocument has methods for creating various elements inside the document, including the CreateElement() and CreateAttribute() methods. It has methods for adding a node to a document: Append(), InsertBefore(), and InsertAfter().

Table 10.2 demonstrates a few key properties and methods of the XmlDocument class.

Note that the CreateElement() and CreateAttribute() methods create the specified structures but do not specifically add them to the document.

Use the XmlNode.AppendNode(), XmlNode.InsertBefore(), or XmlNode.InsertAfter() method to add a node and the XmlElement.SetAttributeNode() method to set an attribute to a node.

Table 10.2: Selected Members of the XmlDocument Class

Member	Type	Description	Example	Example Description
DocumentElement	Property	Sets or gets the root XML element for the document	MessageBox.Show (doc.DocumentElement);	Displays the root element for an XmlDocument named doc
CreateAttribute	Method	Creates a new attribute for an element with the specified name	doc.CreateAttribute (name) ("type");	Creates a new attribute but does not add it
CreateElement (name)	Method	Creates a new element	doc.CreateElement ("question");	Creates a new element but does not add it
Load(filename)	Method	Loads an XML document from the given file name	doc.Load ("myStuff.xml");	Loads the specified file into the document
LoadXml (xmlString)	Method	Interprets a string value as XML	doc.LoadXml ("<simpleXml>this is simple</simpleXml>");	Loads the text value as a simple XML document
Save (filename)	Method	Saves the document to the specified file	doc.Save ("myStuff.xml");	Saves thecurrent document to a file named *myStuff.xml*

Reading an Existing XML Document

The only way to make sense of all this information is to build a program that puts it to use. The XML Viewer program will help you see how exactly C# reads an XML document and how to get around in it. The XML Viewer is more than a programming exercise. It is a tool you can use to analyze and explore the structure of any XML document. After you see how it describes the data in an XML document, you will be ready to get inside the engine and see how the program does its work. Figure 10.7 illustrates the XML Viewer as it first loads the quiz document.

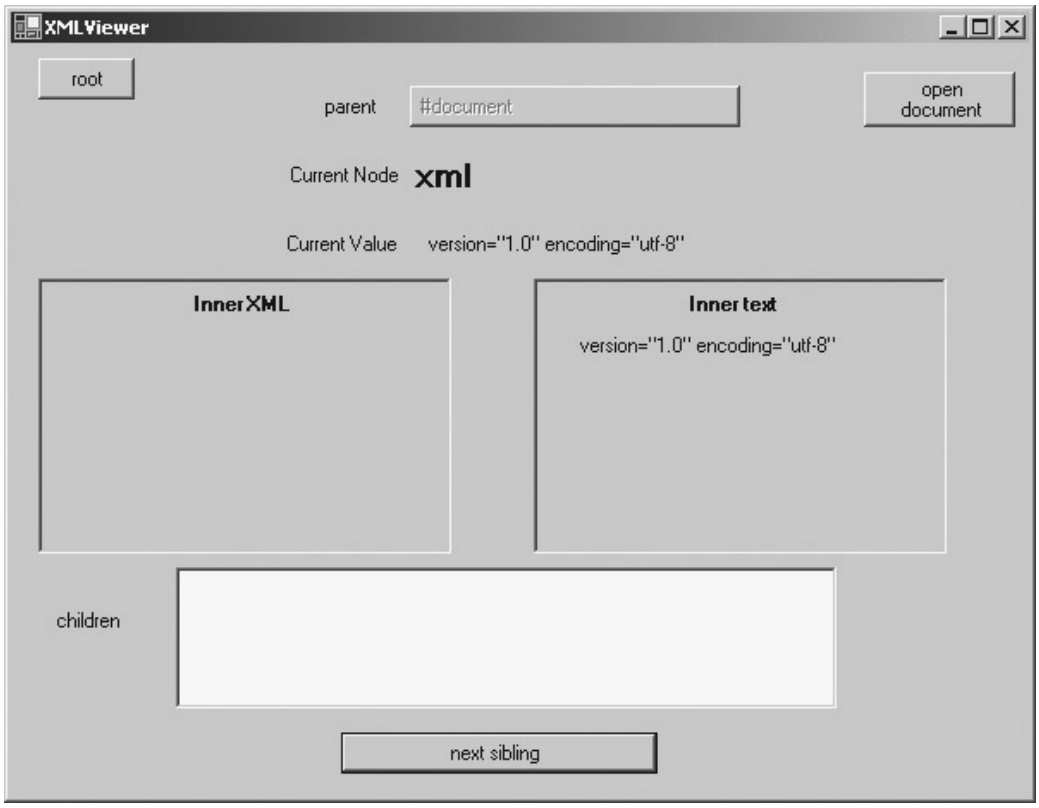


Figure 10.7: The starting node is named *xml*, but it isn't very interesting.

The root node is the xml tag itself. It usually provides some background information, but it doesn't have any children and doesn't contain a lot of other data. To get to the interesting parts of the file, you click the Next Sibling button (see Figure 10.8).

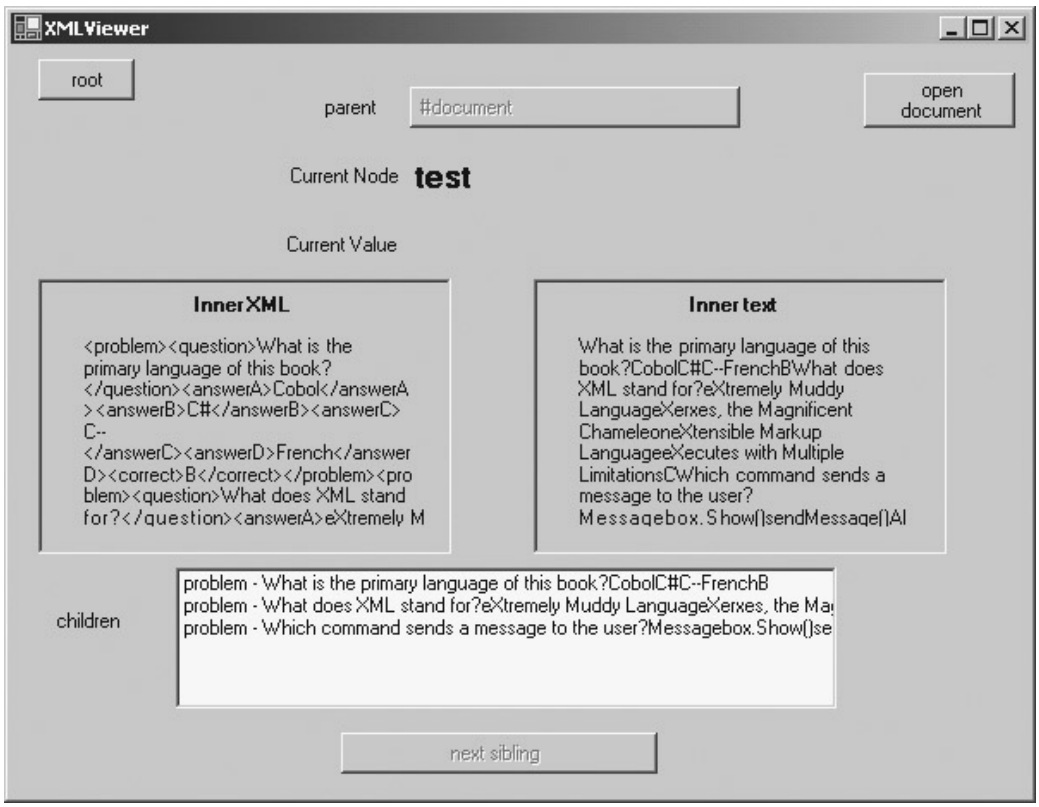


Figure 10.8: The *test* node has a lot more information. (Almost too much!)

The test node is much more interesting. All the interesting data in the quiz file was somehow related to the test node, so it isn't surprising that the test node shows much more information. All the screen elements in the XML Viewer program describe properties of the current node. The Parent button refers to the test node's parent node, which is the document itself. I disabled the Parent button in this particular case because the document is the top layer. The node field refers to the Name property of the current node. In this case, the name of the current node is *test*. The value label refers to the Value property of the current node. You might be surprised that the test node has no value at all. That is because all the information of the test is actually stored in its children. As you will see, the InnerText property often turns out to be more useful than the Value property.

The InnerXml property returns all the XML associated with the current node. When test is the current node, all the XML between the <test> and </test> tags in the original XML show up as the InnerXml property. The InnerText property simply strips out the XML tags from the InnerXml property and returns the resulting text. Notice that the Next Sibling button is disabled because the program has detected that the test element does not have a next sibling.

A node can have several child nodes, as you can see from the test node. In this particular file, the test has three problems, and you can see a list box with three problem nodes. To view one of the child nodes, double–click it. I chose the first problem, which results in Figure 10.9.

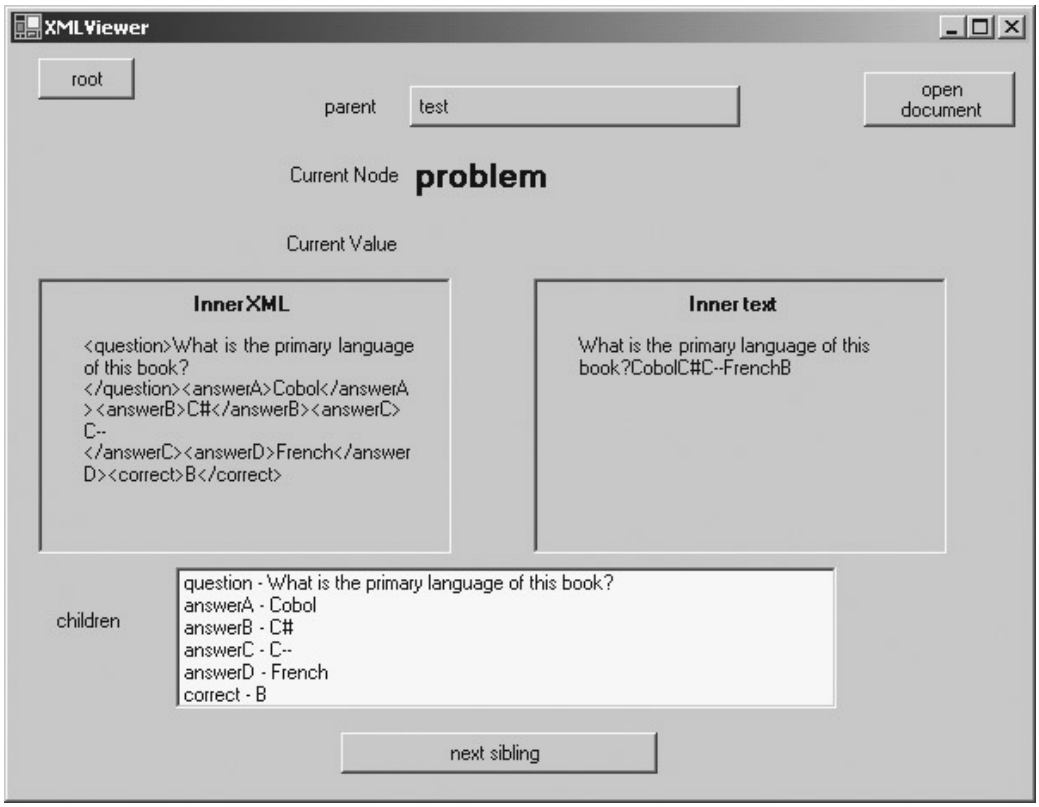


Figure 10.9: I have clicked the first problem, and the viewer is now looking at the problem node.

All the screen elements have changed to reflect that the problem node has the focus. The test node is now listed as the parent node, and you can click the button to return to the parent node. (That's why I made the parent node a button instead of a label.) The problem node still doesn't have a value because it is mainly a container for other nodes. You can see that the InnerXml and InnerText properties have narrowed their scope considerably. Also, the list of ChildNodes shows the question, possible answers, and correct answer. The Next Sibling button is activated because this question has siblings. Click the Next Sibling button a couple times until it becomes disabled. It will do so at the last question, which gives you a screen like Figure 10.10.

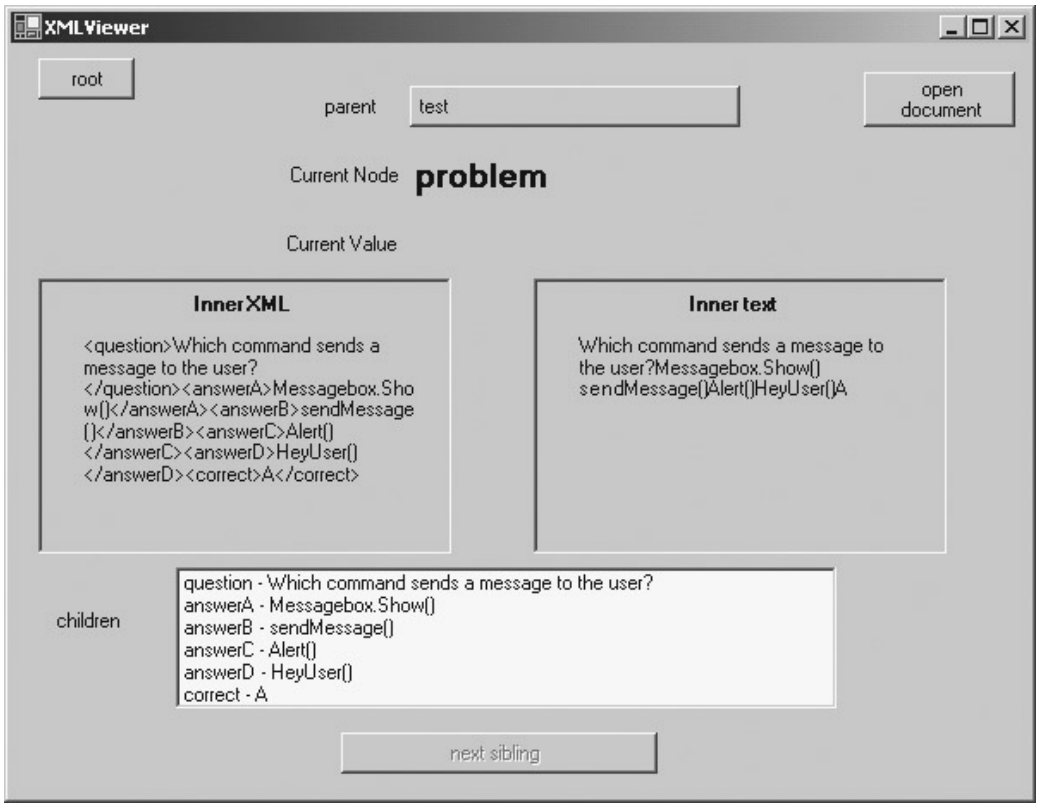


Figure 10.10: Now the viewer is pointed at the last child of the test node.

It's time to investigate the children of the problem node. Click the first child (the question), and you will see something like Figure 10.11.

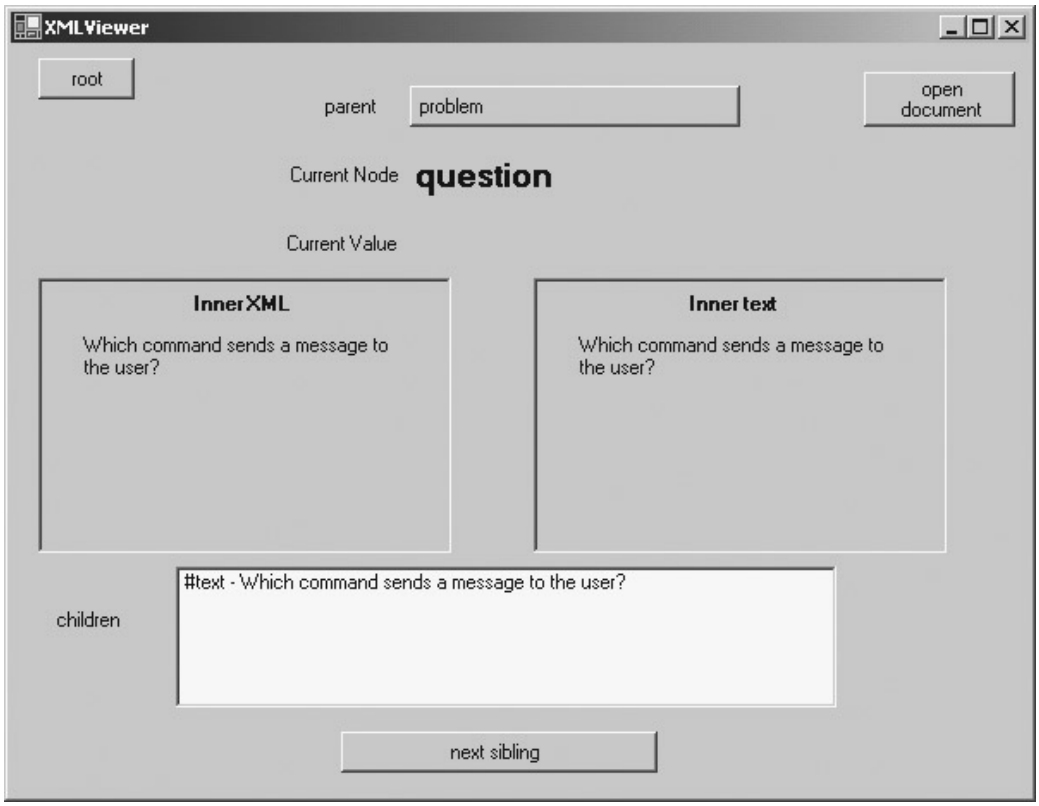


Figure 10.11: You might be surprised that there *still* isn't anything useful in the value field.

It would seem as though the question is the lowest level of the XML document, but it isn't. The InnerText and InnerXml of the question (and all its siblings) are useful information, but you will see

that the question has one child, marked as *#text*. Not until you get to the #text node do you finally have actual data in the value field. The XML Viewer program is particularly interesting because it can be used to analyze *any* legal XML document. This points out one of XML's advantages. The format of the data describes the data's meaning, so a program can work with data in a new format with some hope of deciphering it.

Figure 10.12 illustrates how the .NET framework sees the quiz document.

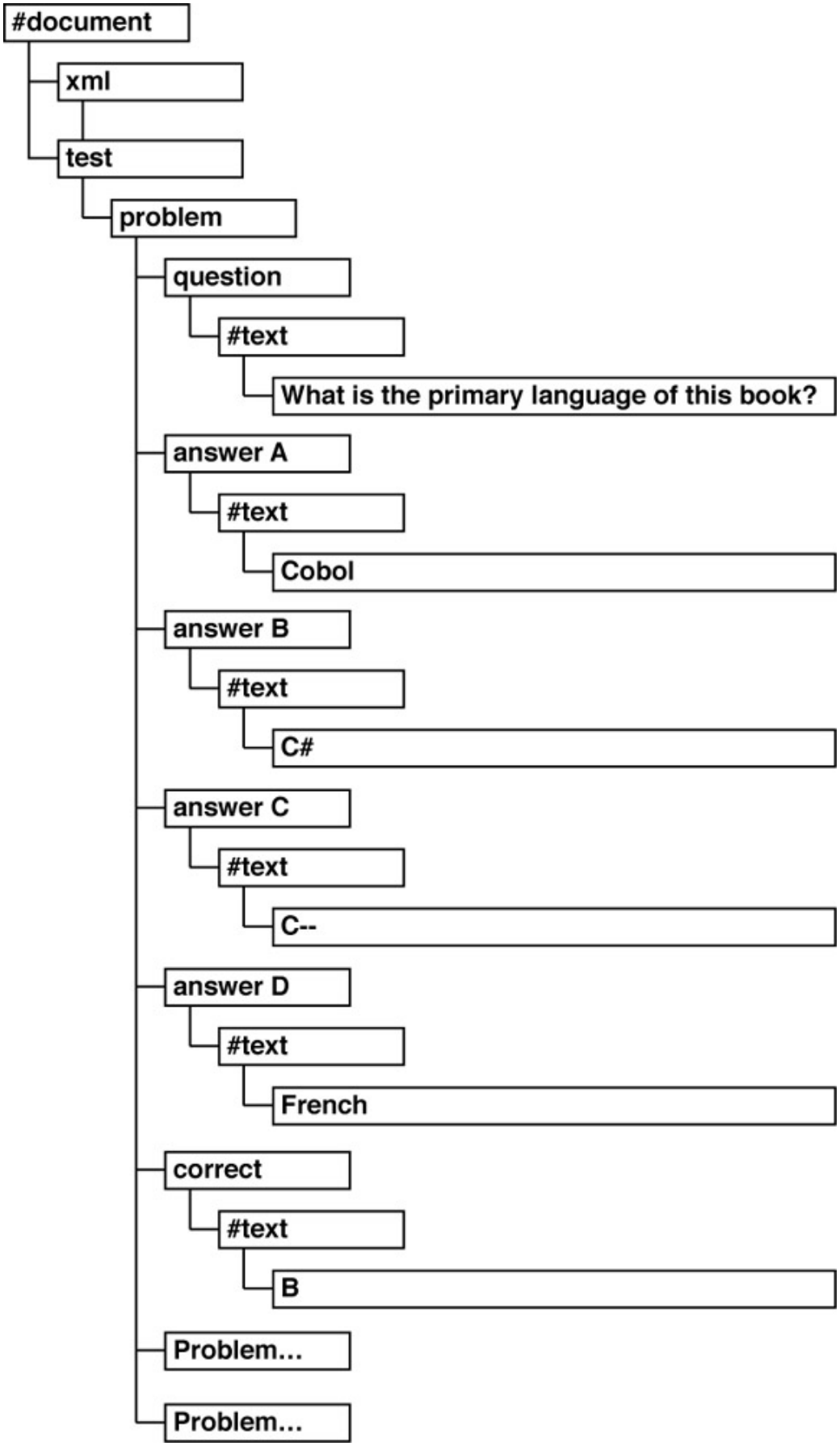


Figure 10.12: This diagram more accurately reflects how .NET sees an XML document.

Trick

It's okay if you're still a little confused, because the various characteristics of a node can be very confusing. If you're still hazy, run the XML Viewer and watch what happens as you move around in a document. Look at the XML source code of your XML document so that you can see how it relates to the various properties of the node object. The best way to make sure that you understand this is to build a quick XML document in the XML editor, load it into the XML Viewer, and see whether you understand how everything fits together. The XML Viewer is really just a window into the XmlNode class, so if you understand how it works, you are well on your way to understanding XML in .NET.

Creating the XML Viewer Program

The design of the XML Viewer program is more transparent than in many programs in this book because the XML Viewer is meant primarily as a programmer's tool. Most of the form's visual elements map directly to a specific property of the XmlNode class.

Building the Visual Layout

The XML Viewer provides a window to a document by providing details about one node at a time. Several properties of the current node are shown, including the Name, Value, InnerText, and InnerXml properties. The values of each of these properties are displayed in an appropriately named label (see Figure 10.13).

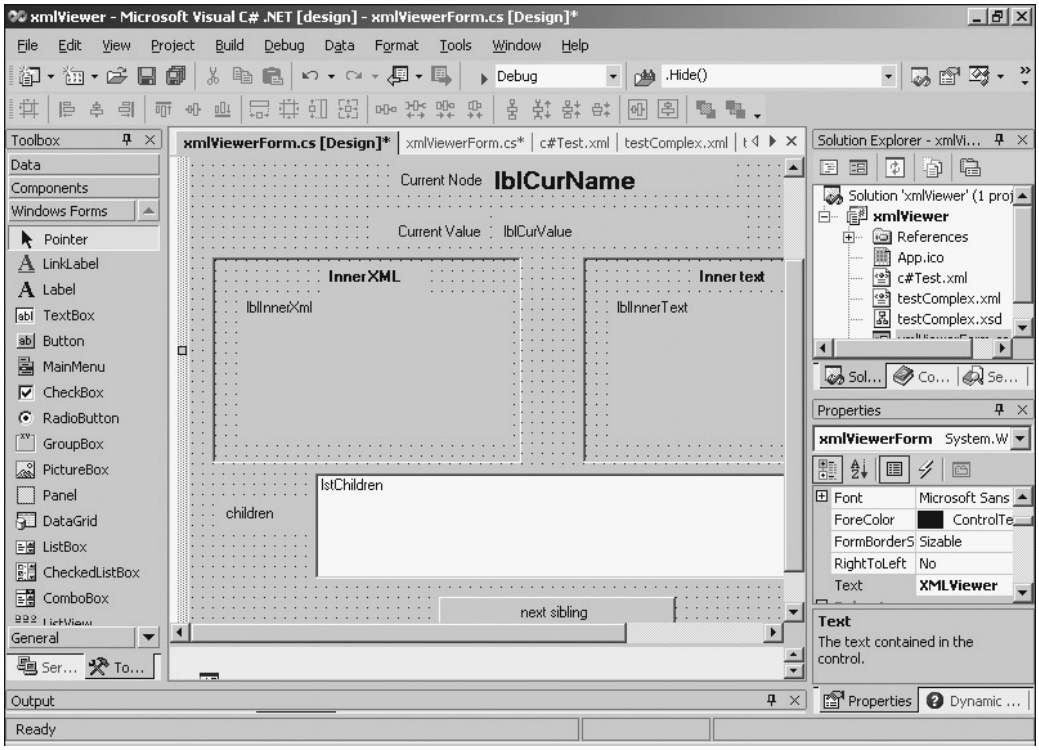


Figure 10.13: Most of the visual interface is composed of labels, with a few buttons and a list box added for navigation.

In addition to the mentioned properties of the node, the XML Viewer displays the name of the parent node in the Parent button. The name and description of each of the current node's child nodes are listed in a text box.

The user can maneuver through the document by clicking the Parent button, double–clicking a child in the list box, clicking the Root button (which takes the user directly to the document's root), or clicking the Next Sibling button, which displays the current node's next sibling, if it has one.

I made the lblInnerText and lblInnerXml labels taller than a typical label because I knew that they would sometimes include large amounts of text. I placed these labels and their descriptors inside panels to break up the screen and make the purpose of the large labels apparent.

I used a list box to display child nodes because it seemed like a natural fit. The children can be any number of elements and can easily be described by string values. The list box seemed like the simplest way to handle the display and selection of child nodes.

Creating Instance Variables

The XML Viewer program has only two form−level variables, and both are classes from the System.Xml namespace. The viewer code begins with a reference to that namespace:

```
using System.Xml;
```

The two instance variables are an XmlDocument object named doc and an XmlNode object named theNode:

```
    private XmlNode theNode;
    private XmlDocument doc;
```

The doc variable will contain a reference to the entire XML document. It will remain the same until the user requests another document with the Open Document button. The theNode variable will change to reflect whichever node is currently being described on the screen.

Initializing the Program

As usual, I did much of my initialization in the form's load event. The main objectives of the xmlReader_Load() method are to load the document from a file, set theNode to the root document of the file, and display the root node:

```
private void xmlReader_Load(object sender,
        System.EventArgs e) {
  doc = new XmlDocument();
  doc.Load("c#Test.xml");
  theNode = doc.FirstChild;
  displayNode();
} // end form load
```

The doc = new XmlDocument() code assigns a new instance of the XmlDocument class to doc. The next line uses doc's Load() method to load a pre−existing file name. The FirstChild property of doc will call the first tag in the document, which is usually the <xml> tag. displayNode() is a method of the XML Reader class that I'll describe in the next section.

Displaying a Node

The code to display a node is the centerpiece of the XML Viewer program. It is involved, so I'll explain the displayMode() method in smaller pieces: (look on the CD−ROM to see the function in its complete form.)

The displayNode() method begins by creating a local XmlNode named childNode. This variable will be used to populate the child list box.

Working with the Plain Text Properties

The first series of statements deals with the properties of a node that return plain text:

```
//handle all the straight text properties
  lblCurName.Text = theNode.Name;
  lblCurValue.Text = theNode.Value;
  lblInnerXml.Text = theNode.InnerXml;
  lblInnerText.Text = theNode.InnerText;
```

There isn't anything at all challenging about this code because all the properties return a text value, which is easily mapped to the Text property of the labels.

Displaying and Enabling the Parent Button

The more interesting work comes when dealing with the parent, child, and sibling properties because they do not simply return a string value, but an actual node. You must take more care when dealing with these properties:

```
  //handle the parent button
  btnParent.Text = theNode.ParentNode.Name;
  //enable btnParent only when appropriate
  if (theNode.ParentNode.NodeType == XmlNodeType.Element){
    btnParent.Enabled = true;
  } else {
    btnParent.Enabled = false;
  } // end if
```

The ParentNode property returns an XmlNode. I mapped the name of this node to a button because I want the user to be able to view the parent by clicking the button. This is easily accomplished, but there's a potential problem. If the user is already at the document's root, the parent is listed as #document, which cannot be traversed. The easiest way to prevent this problem is to check the parent node's type as soon as a node is loaded and to disable the Parent button when displaying the parent node is inappropriate. I used an if statement to compare the NodeType of the parent node to a built–in value called Xml.NodeType.Element. I allowed the Parent button to be displayed only if the parent is an element type. This will be true in all nodes of an XML document but the root, so it prevents moving above the root node.

Trick The code for moving to the parent class belongs in the Parent button's click event. You might think that it would make more sense to put code that traps for an error in the event method where the error will occur (which is in the button click, in this instance). However, sometimes the best kind of error handling is error prevention. Rather than trap for an error after it has occurred (which would happen if you put the code in the button press method), you can prevent the button from being pressed when you know that it will cause an error. Often this type of preventive programming saves you grief in the long run. You will see the practice used in a couple other places in the XML Viewer program.

Enabling the Next Sibling Button

Like the Parent button, the Next Sibling button is enabled only if the program has determined that the current node has a next sibling. If it does not, the NextSibling property returns the value null. Using an else clause is important because it's possible that the button has been disabled by a previous node. Therefore, you must always explicitly set it disabled or enabled, based on the NextSibling property of the current node.

Populating the Children List Box

The child nodes require a different display technique because you cannot predict how many children a node will have. The list box control is a very convenient way to work with child nodes because it is designed to hold lists. Whenever the user wants to display a new node, the list box must be repopulated, based on the current node's children:

```
  //populate the list box with children
  XmlNode childNode;
  lstChildren.Items.Clear();
  if (theNode.HasChildNodes){
    childNode = theNode.FirstChild;
    while (childNode != null){
      lstChildren.Items.Add(childNode.Name
         + " - " + childNode.InnerText);
      childNode = childNode.NextSibling;
    } // end while
  } // end if statement
```

The first task is to clear the list box so that any residual items are removed. Then the program checks whether the current node has any child nodes. If not, there is no need to proceed. If so, the local childNode variable is assigned the first child of the parent node. As long as the child node exists (in other words, isn't null), the name and inner text of the child node are added to the ListBox, and the next sibling is assigned to childNode. If there isn't a next sibling, childNode will get a null value, which ends the loop.

Moving to the Other Nodes

Three buttons on the XML Viewer form are used to move to a new place within the XML document and display the resulting node. The code for these three methods is very similar, so I will present them together:

```
private void btnNextSib_Click(object sender,
    System.EventArgs e) {
  theNode = theNode.NextSibling;
  displayNode();
} // end btnNextSib

private void btnParent_Click(object sender,
        System.EventArgs e) {
  theNode = theNode.ParentNode;
  displayNode();
} // end btnParent

private void btnRoot_Click(object sender,
        System.EventArgs e) {
  theNode = doc.FirstChild;
  displayNode();
} // end btnRoot
```

All three of these buttons set theNode to a new node and then call the displayNode() method to display the new node. The only thing different about the various methods is which node is displayed. The btnNextSib_Click() method sets the button to the current node's next sibling. It isn't necessary to ensure that there is a next sibling here because the displayNode() method disables this button if there are no more siblings. The code can be called only when theNode has a next sibling.

The btnParent_Click() method sets theNode to the current node's parent node. As with the sibling, the showNode() method prevents the Parent button from being active if the current node's parent is not an element.

The btnRoot_Click() method always goes to the root element of the XML document, which is always the first child of the document. There's no need for error checking here because any valid XML document has at least one child element. The first child of the document is commonly referred to as the *root* of the document.

Moving to a Child Node

The child nodes of the current node are stored in the lstChildren list box. The general approach to selecting a child is much like that for the buttons. However, ensuring that the viewer moves to the appropriate node requires attention. First, I did *not* use the default event of the list box because it seemed more appropriate to move to a new child after the user double−clicks a child in the list box. (Usually, if a list box has any direct events, they are related to double clicks. A single click is usually used to change which element in the list box currently has the focus.) The double−click event code uses the selectedIndex property to determine which child the user wants to see and to set theNode appropriately:

```
private void lstChildren_DoubleClick(object sender,
    System.EventArgs e) {
  if (lstChildren.Items.Count > 0){
    theNode = theNode.ChildNodes[lstChildren.SelectedIndex];
    displayNode();
  } // end if
} // end lstChildren
```

Upon testing the program, I discovered that the user could double−click the list box even if the current node has no children, causing an exception. I used an if statement to prevent this situation from occurring.

Opening a New Document

The XML Viewer program works with any valid XML document, not just the one I used as a default. To take advantage of this, I added a button to open a new file. The code for this button displays a File dialog, opens the requested file, and sets the node to the new file's root node:

```
private void btnOpen_Click(object sender,
        System.EventArgs e) {
  if (opener.ShowDialog() != DialogResult.Cancel){
    doc.Load(opener.FileName);
    theNode = doc.FirstChild;
  } // end if
}  // end btnOpen
```

Trick You might want to use the Open button to examine a version of the test XML that might be considered a better design. TestComplex.xml (included on the CD−ROM) has an <answer> element that contains a group of elements. Use the XML Viewer to explore this document and see how it changes things. You might also want to open the XML document in Visual Studio and see how it changes the data mode.

Writing New Values to an XML Document

In addition to reading existing documents, .NET makes creating an XML document from scratch very easy. The XML Creator program illustrates how this can be done. Figures 10.14 and 10.15 show the XML Creator program in action. What makes the XML Creator different from the XML Viewer is that the creator program can create XML without a pre−existing document. The XML Creator emphasizes how to create a new document and new nodes.

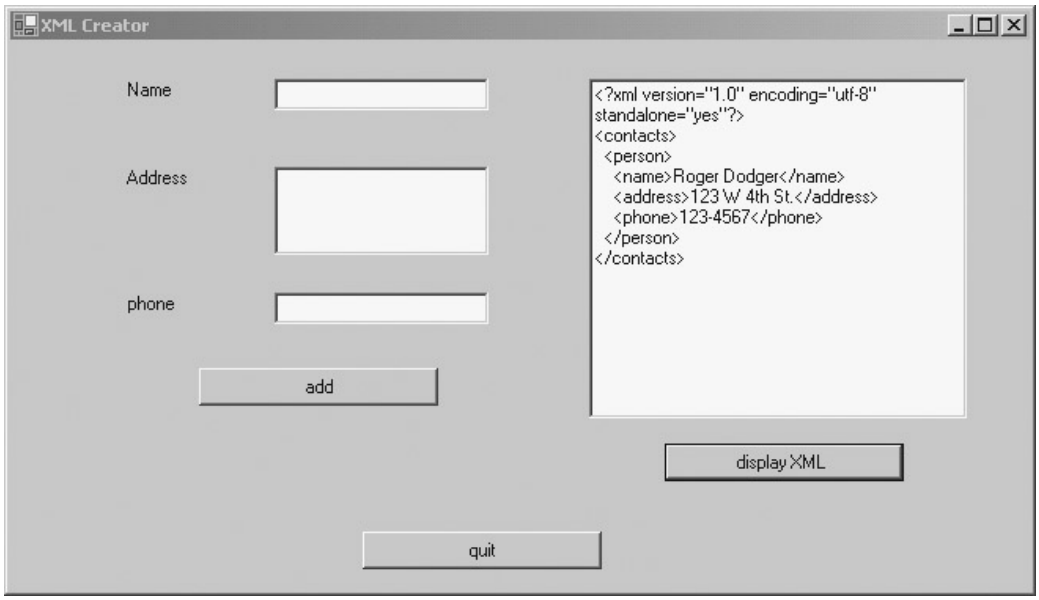


Figure 10.14: The XML displayed in the right panel was created by the program.

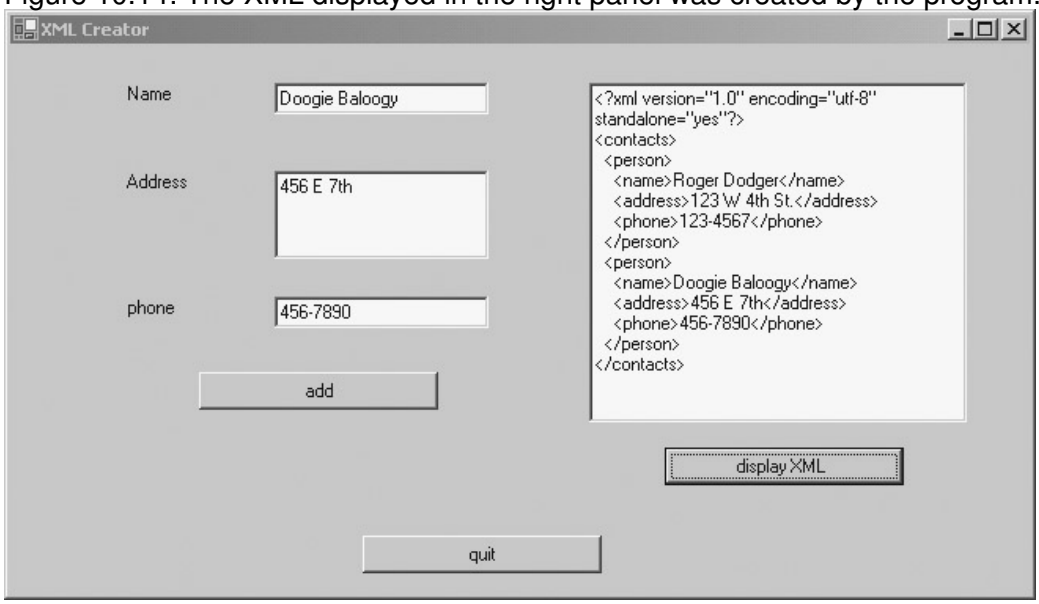


Figure 10.15: After entering a new element in the text boxes, pressing the Add button, and displaying the XML, a new element is visible in the code.

Designating the Class−Level Variables

No new classes are necessary to create an XML document. The program uses an XmlDocument named doc, an XmlNode named theNode, and a string to hold a file name:

```
private XmlDocument doc;
private XmlNode theNode;
private string fileName = "practice.xml";
```

Building the Document Structure

The XML document is created in the load method of the form. Creating the document is not difficult because it is simply a new instance of the XmlDocument class. However, creating the structure of the document takes some thought. For the XML Creator, I decided to build a simple address book. Figure 10.16 shows the basic document structure.

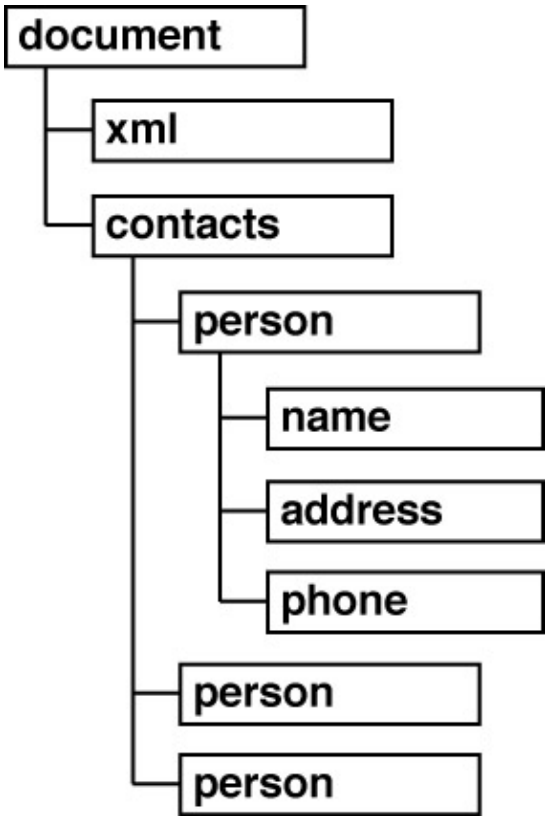


Figure 10.16: The document contains a contact, which is a group of person elements. Each person consists of a name, address, and phone number.

Trick It's a familiar refrain by now, but one that bears repeating. You need to sketch out a diagram like this before you start to code an XML document, or you're going to get confused. Seeing the relationships between elements in your code can be difficult, but a diagram like this clarifies your intentions and makes the code much easier to write.

The code works by creating XmlElement variables for contact and person. theNode will be used as a temporary node. The first element in any document is the root (xml) node. A special method of the XmlDocument object is designed to simplify the creation of a root node: CreateXmlDeclaration() makes a root node. The parameters are almost always as listed. The first parameter stands for the XML type, which should always be "1.0". The second parameter stands for the encoding, which should always be "utf–8". The last parameter describes whether the document can stand alone. It should be "yes". After you create the root node, you still have to add it to the document. Use the AppendChild() method of the XmlNode class to add a child to a node (or to the document itself, which is also a node).

To create the other elements, use the CreateElement() method of the XmlDocument class. You will still need to use the AppendChild() method of whichever node you want to add the new child to. For example, the following code creates a node named *contacts* and appends it to the document object:

```
//create contacts node
contacts = doc.CreateElement("contacts");
doc.AppendChild(contacts);
```

The next code fragment creates a person node and adds it as a child to contacts:

```
  //create first address
  person = doc.CreateElement("person");
  contacts.AppendChild(person);
```

When the basic structure is in place, it's time to add some elements to the person class. Each of these elements is created just like contacts and person, but the other elements hold actual data. The easiest way to add information to a node is to set the InnerText property of the node in question. For example, here's how I set up the name element:

```
  theNode = doc.CreateElement("name");
  theNode.InnerText = "Roger Dodger";
  person.AppendChild(theNode);
```

The entire code for the form's load event shows how the entire document is designed:

```
private void XmlCreatorForm_Load(object sender,

    System.EventArgs e) {
  //initialize
  doc = new XmlDocument();

  XmlElement contacts;
  XmlElement person;

  //create root node
  theNode = doc.CreateXmlDeclaration("1.0", "utf-8", "yes");
  doc.AppendChild(theNode);

  //create contacts node
  contacts = doc.CreateElement("contacts");
  doc.AppendChild(contacts);

  //create first address
  person = doc.CreateElement("person");
  contacts.AppendChild(person);

  //create address elements
  theNode = doc.CreateElement("name");
  theNode.InnerText = "Roger Dodger";
  person.AppendChild(theNode);

  theNode = doc.CreateElement("address");
  theNode.InnerText = "123 W 4th St.";
  person.AppendChild(theNode);

  theNode = doc.CreateElement("phone");
  theNode.InnerText = "123-4567";
  person.AppendChild(theNode);
} // end form load
```

As you can see from the code, it's possible to create an XML document entirely from scratch, but you must have a solid idea of the document's structure.

Adding an Element to the Document

You can create additional elements in the same way you create the first one. However, after the basic structure of an element is defined, it's much easier to make a copy of an existing element than

to build one from scratch.

The code in btnAdd's click event illustrates how this is done:

```
private void btnAdd_Click(object sender, System.EventArgs e) {
  //duplicate the person node
  XmlNode contacts;
  XmlNode person;
  XmlNode root;

  root = doc.FirstChild;
  contacts = root.NextSibling;
  person = contacts.FirstChild;
  theNode = person.Clone();

  //copy node values from text boxes
  theNode["name"].InnerText = txtName.Text;
  theNode["address"].InnerText = txtAddress.Text;
  theNode["phone"].InnerText = txtPhone.Text;

  //add the new node to contacts
  contacts.AppendChild(theNode);

} // end btnAdd
```

The first part of the code recreates the document's structure by extracting the root node, the contacts node, and the person node. Next, I created a copy of the person node, named theNode, by using the Clone() method of the person node.

I then copied the values from the text boxes over to the new node. Notice how you can use the node's name inside braces to specify which node you want to work with.

Finally, I added the new node to contacts.

Trap Keep your diagram handy as you're adding nodes. The document structure expects all person nodes to be added to the contacts node. You will get unpredictable results if you add the person node somewhere else.

Displaying the XML Code

To see that something is happening, I decided to display the code in the text box. In principle, this is very easy to do. Every node (including doc) has an OuterXml property that returns the XML code of the node and all its children (the InnerXml property returns the XML code of the node's children, but not the node itself). However, the formatting that is useful for human readers (with carriage returns and indentation) is usually stripped out in the internal representations of XML code because it can confuse the *parser* (the part of the program that navigates the XML structure). Figure 10.17 illustrates what happens when you simply copy the internal XML to the text box.

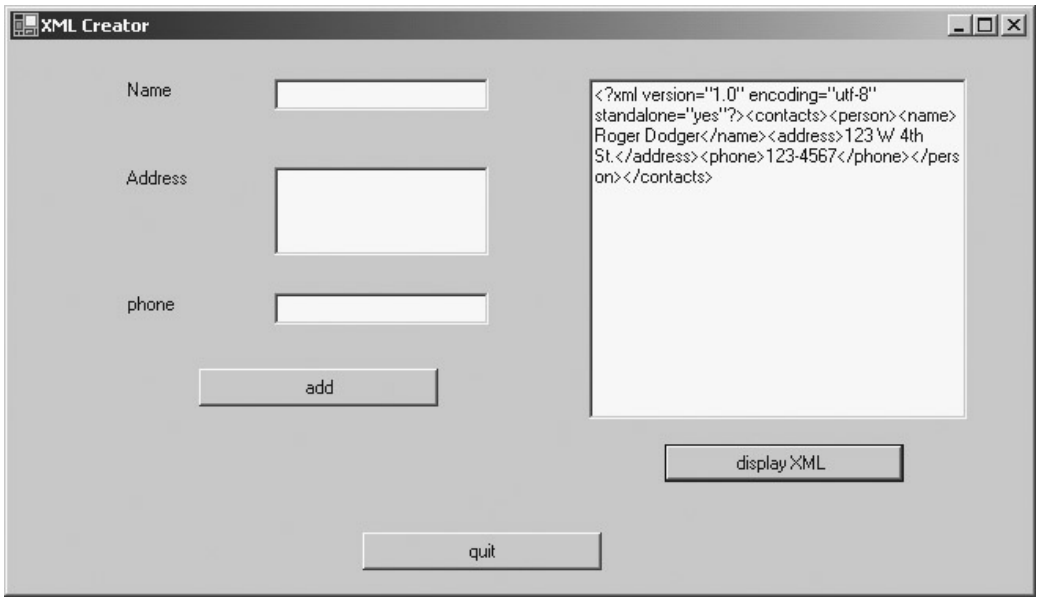


Figure 10.17: The native XML code is complete but very difficult for humans to read.

Fortunately, the XmlDocument class has a property named PreserveWhiteSpace. When this property is set to true, the document is formatted for human reading. When the property is set to false, all whitespace (carriage returns and space characters) are removed. To get the effects of the PreserveWhiteSpace property, you need to reload the document.

The btnDisplay click event takes advantage of the PreserveWhiteSpace property to display the code in a human–readable format and then convert it back to the compressed form preferred by .NET:

```
private void btnDisplay_Click(object sender,
        System.EventArgs e) {
  //save the current document
  doc.Save(fileName);

  //display with whitespace
  doc.PreserveWhitespace = true;
  doc.Load(fileName);
  txtOutput.Text = doc.OuterXml;

  //reload without whitespace
  doc.PreserveWhitespace = false;
  doc.Load(fileName);

} // end btnDisplay
```

I started by saving the XML file as it currently exists to whatever file is determined by the fileName variable (set at the beginning of the program). I then set PreserveWhitespace to true, reloaded the document, and displayed the formatted version in the text box. The formatted version seemed to cause problems for some of the XML code, so I turned off PreserveWhitespace and reloaded the document again so that the version of doc in memory has no whitespace. There are other ways to automate the formatting of an XML document, but this appears to be the simplest.

Examining the Quizzer Program

As usual, the final program for the chapter does not introduce any new code or ideas. The Quizzer game simply puts together the concepts you have learned into an interesting package. I also borrowed heavily from the Adventure Kit game in Chapter 9 because the underlying structure is

quite similar. The program has a main menu screen, which calls an editor and a quiz program.

Building the Main Form

The main form (XmlMainForm) is the simplest form of the project. It consists of three buttons, which call the other forms or exit the program altogether.

Creating Instance Variables

XmlMainForm has only two instance variables. Both are instances of the other two forms in the project:

```
private frmQuiz theQuiz;
private frmEdit theEditor;
```

These variables will be used to create the other forms when they are needed.

Creating the Visual Design of the Form

The visual design of XmlMainForm is quite simple. It has three buttons. The first two buttons call other forms, and the last one exits the program (see Figure 10.18).

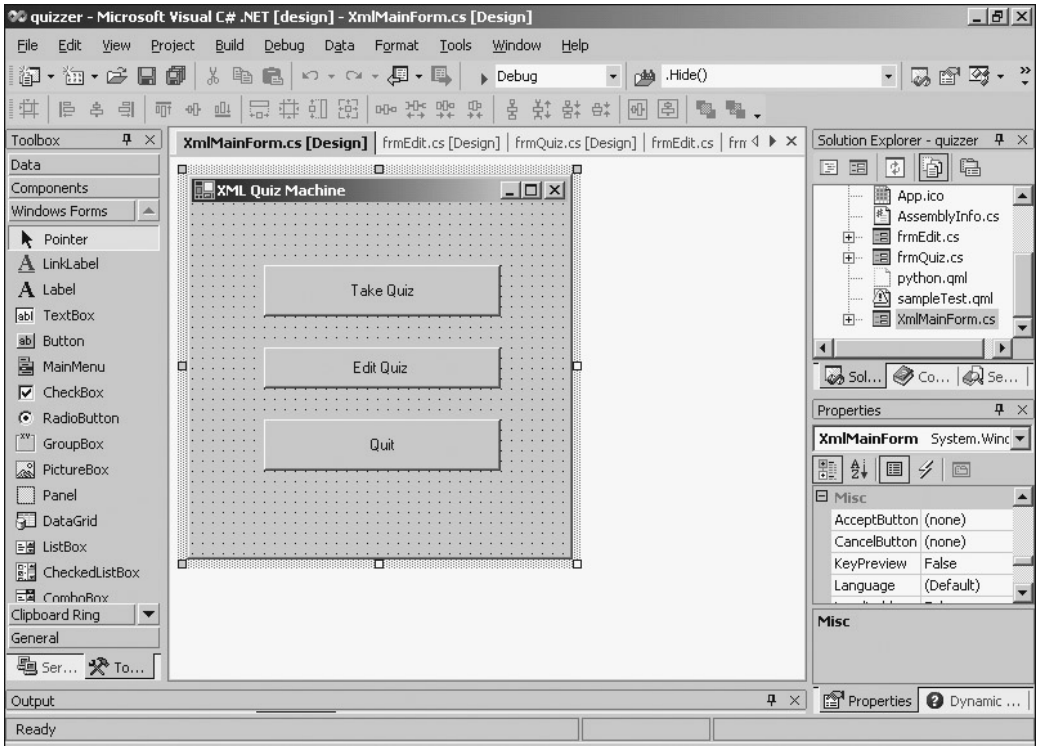


Figure 10.18: As usual, the layout of the main form is extremely simple.

Responding to the Button Events

All the important work is encapsulated in the various other forms, so the main buttons call these forms to do their work:

```
private void btnTake_Click(object sender,
        System.EventArgs e) {
  theQuiz = new frmQuiz();
  theQuiz.Show();
```

```
} // end btnTake

private void btnEdit_Click(object sender,
        System.EventArgs e) {
  theEditor = new frmEdit();
  theEditor.Show();
} // end btnEdit

private void btnQuit_Click(object sender,
        System.EventArgs e) {
  Application.Exit();
} // end btnQuit
```

The Quit button uses the Application.Exit() method to close the entire application.

Writing the Quiz Form

The quiz form is responsible for displaying the quiz. It examines an XML document and copies the appropriate information to the form elements. It also has the capability to navigate to other questions and grade the quiz.

Designing the Visual Layout

The quiz form is designed to show one problem at a time. The question goes in a label at the top of the screen, and the various answers are placed in radio buttons (also called *option buttons*).

In the Real World

Of all the visual design elements, radio buttons seem to have the most inconsistent naming convention. Some languages call them *radio buttons*, some call them *option buttons*, and some call them *grouped check boxes*. C# uses the term *radio buttons*, but many programmers still call them *option buttons* because that's what they were called in Visual Basic. It doesn't matter how you refer to them in your own code, as long as you're consistent. However, if you're working on a professional project with a group of programmers, you will probably be required to follow a standard naming convention.

Two buttons enable navigation forward and backward through the quiz, and a label indicates which question is currently being displayed.

The form also features a small menu structure. The File menu has Open, Exit, and Grade options. Figure 10.19 shows the form in the designer.

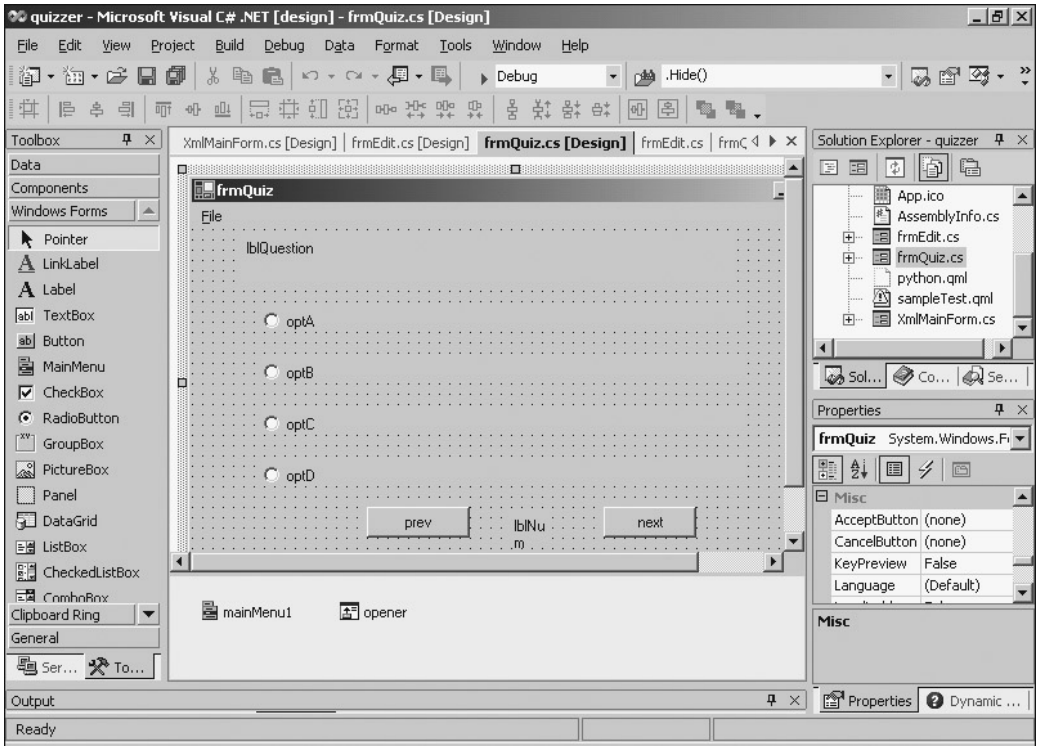


Figure 10.19: The FrmQuiz form has a label for the question and radio buttons for the answers

Creating Instance Variables

The instance variables for the quiz form are used primarily to examine the underlying XML document:

```
private XmlDocument doc;
private XmlNode theTest;
private int qNum = 0;
private int numQuestions = 0;
private string[] response;
```

I created XmlDocument and XmlNode variables to hold the document and the various elements. The qNum variable holds the question number. numQuestions holds the number of questions, and response is a string array to hold the user's responses to the questions.

Initializing in the Load Event

In the form's load event, I created a new XmlDocument and loaded the sample test into it. The code runs more smoothly if there is always an XML document loaded, so I forced the sample document to be loaded as a default. Of course, the user will be able to load any other quiz he or she wants. The resetQuiz() method (described in the next section) will initialize the quiz and display the first question:

```
private void frmQuiz_Load(object sender,
        System.EventArgs e) {
  doc = new XmlDocument();
  doc.Load("sampleTest.qml");
  resetQuiz();
} // end Load
```

Resetting the Quiz

The program needs to initialize a quiz a couple times (generally after a new quiz has been loaded). The resetQuiz() method prepares the quiz:

```
private void resetQuiz(){
  theTest = doc.ChildNodes[1];
  numQuestions = theTest.ChildNodes.Count;
  response = new String[numQuestions];
  for (int i = 0; i < numQuestions; i++){
    response[i] = "X";
  } // end for loop
  qNum = 0;
  showQuestion(0);
} // end resetQuiz
```

I began by assigning theTest the first child of the document. The program can then determine the number of questions by accessing theTest's ChildNodes property. ChildNodes is a collection, so it has a Count property.

When the user begins a new quiz, it is also necessary to initialize the response array. This array will hold all the user responses so that the quiz can be graded. I set the initial value of each response to "X" to indicate that the question has not yet been answered. This way, if a user gets a question incorrect, it will be possible to tell whether he or she responded at all. (I didn't take advantage of this feature in this version of the quiz program, but it doesn't hurt to set up things this way.) I set qNum to 0 and showed question zero using the showQuestion() method.

Moving to the Preceding Question

The user will navigate through the quiz by clicking buttons. The Prev button moves backward one element in the document, stores the user's response with the getResponse() method, checks to ensure that the user has not passed the beginning of the document, and displays the appropriate node using the showQuestion() method:

```
private void btnPrev_Click(object sender,
        System.EventArgs e) {
  getResponse();
  qNum--;
  if (qNum < 0){
    MessageBox.Show("First Question");
    qNum = 0;
  } else {
    showQuestion(qNum);
  } // end if
} // end btnPrev
```

Moving to the Next Question

The code for the Next button is very similar to the code for the Prev button, except that the logic is different when the user reaches the last item in the quiz:

```
private void btnNext_Click(object sender,
        System.EventArgs e) {
  getResponse();
  qNum++;
  if (qNum >= numQuestions){
    qNum = theTest.ChildNodes.Count -1;
```

```
    if (MessageBox.Show("Last Question. Grade Quiz?",
        "Last Question",
        MessageBoxButtons.YesNo,
        MessageBoxIcon.Question) == DialogResult.Yes){
      gradeTest();
    } // end if
  } else {
    showQuestion(qNum);
  } // end if
} // end btnNext
```

If the user has moved beyond the last item, the program resets the question number to the last question and informs the user of the situation. The method uses a message box to ask whether the user wants to grade the program. (It is logical that the user might want a grade at the end of the quiz, but it's also possible that he or she will want to back up and review his or her answers first.)

If the user responds yes to the message box, the gradeTest() method calculates the user's score. If the user is not past the end of the quiz, the showQuestion() method displays the new question.

Checking the Radio Buttons for a Response

The user indicates his or her responses to the questions by choosing one of the radio buttons. The program needs to store this information so that the grading method will be able to compare each user response to the corresponding correct answer. The easiest way to store the responses is with an array. The response array stores one string for each question in the quiz. The btnNext() and btnPrev() methods call this method before the user moves away from a question, so the response array should always indicate all the user's responses (or an "X" if the user has not yet responded to a particular question).

```
private void getResponse(){
  //queries the buttons for user input, copies to response array
  if (optA.Checked){
    response[qNum] = "A";
  } else if (optB.Checked){
    response[qNum] = "B";
  } else if (optC.Checked){
    response[qNum] = "C";
  } else if (optD.Checked){
    response[qNum] = "D";
  } else {
    response[qNum] = "X";
  } // end if
} // end get response
```

The getResponse() method simply looks at all the option buttons to see which one is checked and sets the response element that corresponds with the current question to a value indicating the user's response. If the user has not clicked any option buttons, the current response element will get the value "X".

Displaying a Question

The showQuestion() method is the workhorse of the quiz form. It accepts a question number as a parameter, extracts the appropriate question from the XML document, and displays the question on the form:

```
private void showQuestion(int qNum){
  theTest = doc.ChildNodes[1];
```

```
  XmlNode theProblem = theTest.ChildNodes[qNum];
  lblQuestion.Text = theProblem["question"].InnerText;
  optA.Text = theProblem["answerA"].InnerText;
  optB.Text = theProblem["answerB"].InnerText;
  optC.Text = theProblem["answerC"].InnerText;
  optD.Text = theProblem["answerD"].InnerText;

  lblNum.Text = Convert.ToString(qNum);

  //clear up the checkBoxes
  optA.Checked = false;
  optB.Checked = false;
  optC.Checked = false;
  optD.Checked = false;

  //indicate response if there is one
  if (response[qNum] == "A"){
    optA.Checked = true;
  } else if (response[qNum] == "B"){
    optB.Checked = true;
  } else if (response[qNum] == "C"){
    optC.Checked = true;
  } else if (response[qNum] == "D"){
    optD.Checked = true;
  } else {
    //MessageBox.Show("There's a problem!");
  } // end if

} // end showQuestion
```

The first part of the method assigns doc.ChildNodes[1] to a variable named theTest and the current problem (theTest.ChildNodes[qNum]) to a node named theProblem.

In the Real World

You might wonder how I knew that the test element would be doc.ChildNodes[1] and where the problem would be stored. In the quiz program, I'm using a custom form of XML I designed, so I know exactly where everything should be. My program will create the XML, so it should be in exactly the right format. Even when you know the document structure, figuring out exactly how it looks to the .NET parser can be challenging. When you create your own XML scheme, you might want to examine it in the XML Viewer program presented earlier in this chapter so that you can be sure that you know how the internal document structure is organized.

When I have a reference to the current problem in theProblem, it's easy to copy the inner text of theProblem's nodes to the appropriate labels and option buttons. I also put the current question number in lblNum. The question number is handy for the user, but I really put lblNum in as a debugging tool to ensure that the Next and Prev buttons were working correctly. Examining the question number is much easier than trying to remember which question is the first or last question.

Determining which radio button (if any) should be checked requires thought because, in the quiz form, this is determined by the user's response, which is not stored in the XML document at all, but in the response array. I started by setting the Checked property of each check box to false. Then I examined the response element corresponding to the current question, turning on the Checked property of the corresponding radio button.

Grading the Test

The quiz isn't very interesting without feedback. The gradeTest() method compares the response array to the correct element in each problem of the test XML:

```
private void gradeTest(){
  int score = 0;
  for (int i = 0; i < numQuestions; i++){
    XmlNode theProblem = theTest.ChildNodes[i];
    string correct = theProblem["correct"].InnerText;
    if (response[i] == correct){
      score ++;
    } // end if
  } // end for loop
  MessageBox.Show("Score: " + score + " of " + numQuestions);
} // end gradeTest
```

The score variable holds the current score. For each question in the quiz, I extract the inner text of the correct node and compare this against the corresponding value of response. If the two values are equal, the user got the right answer, and the score should be incremented. At the end of the test, I display the score.

In the Real World

The gradeTest() method would be an ideal place to extend the program's capabilities. For example, you might want to create another XML document to track all the users who have taken the quiz and store their results. You might also want to do more detailed analysis of a user's score, such as which questions the user missed and which questions the user simply didn't answer. You might also want to screen for unanswered questions and allow the user to go back without grading the quiz if there are unanswered questions.

Opening a Quiz

The File menu includes a command for opening a new quiz. This is done just like opening any other XML document:

```
private void mnuOpen_Click(object sender,
        System.EventArgs e) {
  if (opener.ShowDialog() != DialogResult.Cancel){
    doc.Load(opener.FileName);
    resetQuiz();
  } // end if
} // end mnuOpen
```

I displayed a File Open dialog and used the resulting file name to open the document using the doc.Load() method.

After a document is loaded into memory, it is necessary to reset the quiz, which I did with the resetQuiz() method.

Responding to Other Menu Requests

The other two menu events require quite simple code:

```
private void mnuGradeQuiz_Click(object sender,
        System.EventArgs e) {
  gradeTest();
} // end gradeTest

private void mnuExit_Click(object sender,
    System.EventArgs e) {
  this.Close();
} // end mnuExit
```

The Grade Quiz menu calls the gradeTest() method, and the Exit menu closes the current form.

Writing the Editor Form

The editor form was designed to be parallel to the quiz form in its general structure. The editor's main purpose is to allow the user to generate new quizzes, rather than to display existing quizzes. For this reason, the editor uses many features described in the XML Creator program to create new documents and nodes and populate these nodes with values from the form.

Designing the Visual Interface

Figure 10.20 shows the editor's visual interface. I tried to keep the visual interface as similar to the quiz form as possible, but I used text boxes for user input. The editor has four radio buttons (named *optA—optD*), but I sized the radio buttons so that their text attributes would not be visible. I carefully placed the text boxes where the radio buttons' text would normally go. This gives the effect of an editable radio button. The buttons and label at the bottom of the form are just like those in the quiz form. The program has two menus containing elements to load and store the quiz, create a new quiz, add a new question, and close the editor.

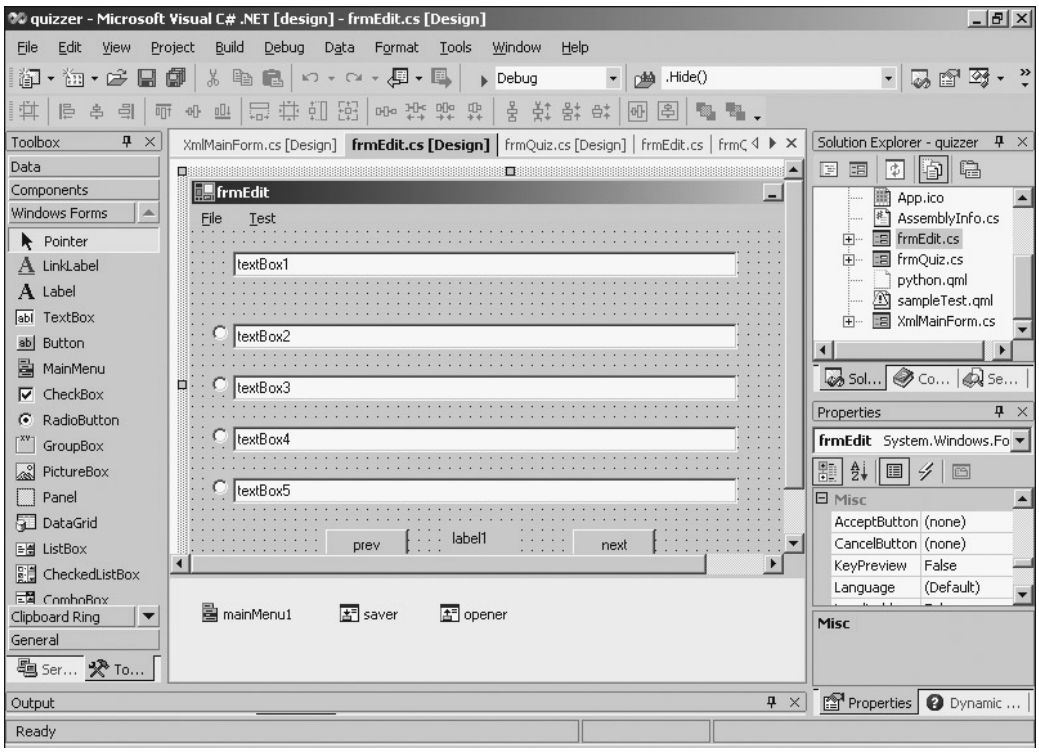


Figure 10.20: The editor form relies on text boxes to display and retrieve the problems.

Creating the Instance Variables

The instance variables for the editor are typical for the programs in this chapter:

```
private XmlDocument doc;
private XmlNode theTest;
private int qNum = 0;
private int numQuestions = 0;
```

The doc variable will hold the entire quiz document, and theTest will hold a reference to the test. qNum is the current question number, and numQuestions holds the total number of questions in the quiz.

Initializing in the Load Event

The form's load event loads a sample test to ensure that an XML document is always in memory. As in the quiz program, this eliminates the need for certain kinds of error checking, and the user is free to modify the default quiz, load another quiz, or create a new one.

```
private void frmEdit_Load(object sender,
        System.EventArgs e) {
  //load up a sample test.
  doc = new XmlDocument();
  doc.Load("sampleTest.qml");
  resetQuiz();

} // end frmEdit_Load
```

The frmEdit_Load() method simply loads up a default document and calls the resetQuiz() method (described in the next section) to start the quiz editing process.

> **Trick** Because I'm using a specific style of XML markup, I decided to give it its own extension (*qml* for *quiz markup language*). You commonly do this when you're working with a specific document structure. You use the more generic xml extension when the exact structure of the document isn't as important (as in the XML Viewer program, which was designed to handle any XML document).

Resetting the Quiz

Resetting the quiz works exactly the same way in the editor as in the quiz program:

```
private void resetQuiz(){
  theTest = doc.ChildNodes[1];
  numQuestions = theTest.ChildNodes.Count;
  qNum = 0;
  showQuestion(0);
} // end resetQuiz
```

The test is retrieved from doc.ChildNodes[1], and numQuestions extracts the number of questions from the test object.

I reset qNum to 0 and showed the initial question using the showQuestion() method.

Showing a Question

Showing a question in the editor is much like showing it in the quiz program, except that the radio button values are extracted from the XML document in the editor, rather than in the response array in the quiz program:

```
private void showQuestion(int qNum){
  XmlNode theProblem = theTest.ChildNodes[qNum];
  txtQuestion.Text = theProblem["question"].InnerText;
  txtA.Text = theProblem["answerA"].InnerText;
  txtB.Text = theProblem["answerB"].InnerText;
  txtC.Text = theProblem["answerC"].InnerText;
  txtD.Text = theProblem["answerD"].InnerText;
  lblNum.Text = Convert.ToString(qNum);

  //uncheck all the option buttons
  optA.Checked = false;
  optB.Checked = false;
  optC.Checked = false;
  optD.Checked = false;

  //Check the appropriate option button
  switch (theProblem["correct"].InnerText){
    case "A":
      optA.Checked = true;
      break;
    case "B":
      optB.Checked = true;
      break;
    case "C":
      optC.Checked = true;
      break;
    case "D":
      optD.Checked = true;
      break;
    default:
      // do nothing
      break;
  } // end switch
} // end showQuestion
```

The radio buttons are set by extracting the correct element from the current problem and setting the Checked property of the corresponding radio button.

Updating a Question

Whenever the user moves to a new question, the program stores the current question's data to the internal XML structure by copying the values of the appropriate text boxes to the current problem node:

```
private void updateQuestion(int qNum){
  // updates the current question's XML
  XmlNode theProblem = theTest.ChildNodes[qNum];
  theProblem["question"].InnerText = txtQuestion.Text;
  theProblem["answerA"].InnerText = txtA.Text;
  theProblem["answerB"].InnerText = txtB.Text;
  theProblem["answerC"].InnerText = txtC.Text;
  theProblem["answerD"].InnerText = txtD.Text;

  //store the correct answer based on the option buttons
```

```
  if (optA.Checked){
    theProblem["correct"].InnerText = "A";
  } else if (optB.Checked){
    theProblem["correct"].InnerText = "B";
  } else if (optC.Checked){
    theProblem["correct"].InnerText = "C";
  } else if (optD.Checked){
    theProblem["correct"].InnerText = "D";
  } else {
    theProblem["correct"].InnerText = "X";
  } // end if
} // end updateQuestion
```

The correct value cannot be directly determined from a text box entry, so it is generated by evaluating which radio button has been checked.

Trick It might seem that the correct and response elements are a pain to work with compared to the other elements. True, the radio buttons require more attention than the text elements. This effort is worth it in the long run, however. Most of the elements in a problem are simply text and don't require any error checking. If you let the user type in an answer, it would be easier to copy the values to and from the resulting text box, but you would have to do all kinds of validation. You would have to ensure that the user typed a legal response, used the correct case, and didn't try to type the entire answer instead of the letter corresponding to the answer. The overhead associated with using radio buttons for input is offset by the knowledge that the user input will always fall within a predictable range.

Moving to the Preceding Question

Moving to the preceding question works almost exactly the same in the editor and the quiz program. The only new wrinkle is that the editor does not allow the user to move on without clicking one of the option buttons. This ensures that every question will have a response. Otherwise, the quiz could have questions that would be impossible to answer. The noResponse() method (described in the next section) returns the boolean value true if there are no responses and false if at least one of the check boxes has been selected. If no responses are selected, the code reminds the user that something must be selected. If a radio button has been selected, the code proceeds to update the current question, decrement the question number, check to ensure that the question number isn't less than 0, and display the new question.

```
private void btnPrev_Click(object sender,
        System.EventArgs e) {
  if (noResponse()){
    MessageBox.Show("You must select one of the answers");
  } else {
    updateQuestion(qNum);
    qNum--;
    if (qNum < 0){
      qNum = 0;
      MessageBox.Show("First question");
    } else {
      showQuestion(qNum);
    } // end 'first question' if
  } // end answerEmpty if
 } // end btnPrev
```

Moving to the Next Question

The next question code is similar to the preceding question code. However, when the user reaches the last question in the editor, the code prompts to see whether the user wants to add a new question. In an editor, it's possible that the user will want to add a new question at the end of the quiz.

```
private void btnNext_Click(object sender,
         System.EventArgs e) {

  if (answerEmpty()){
    MessageBox.Show("You must select one of the answers");
  } else {
    updateQuestion(qNum);
    qNum++;
    if (qNum >= numQuestions){
      qNum = numQuestions -1;
      if (MessageBox.Show("Last question. Add new question?",
          "last question",
          MessageBoxButtons.YesNo,
          MessageBoxIcon.Question) == DialogResult.Yes){
        mnuAddQuestion_Click(sender, e);
      } // end 'add question' if
    } else {
      showQuestion(qNum);
    } // end 'last question' if
  } // end 'answer empty' if

}  // end btnNext
```

Checking for a No Response

The btnNext and btnPrev events need to know whether the option buttons are *empty* (that is, none of the responses have been checked). This is done by setting a boolean variable named responseEmpty to true. Then I checked each option button to see whether it was checked. If so, responseEmpty is set to false. I then returned the value of responseEmpty. If any radio button has been selected, the noResponse() method returns a value of false. If none of the radio buttons have been selected, noResponse() returns the value true.

```
private bool noResponse(){
  //checks to see if all the check boxes are empty
  bool responseEmpty = true;
  if (optA.Checked){
    responseEmpty = false;
  } // end if
  if (optB.Checked){
    responseEmpty = false;
  } // end if
  if (optC.Checked){
    responseEmpty = false;
  } // end if
  if (optD.Checked){
    responseEmpty = false;
  } // end if
  return responseEmpty;
}// end noResponse
```

Saving a File

Saving the file is a simple affair:

```
private void mnuSaveAs_Click(object sender,
        System.EventArgs e) {
  if (saver.ShowDialog() != DialogResult.Cancel){
    doc.Save(saver.FileName);
  } // end if
} // end mnuSave
```

I show a File Save dialog box and use it to get the user's requested file name. I then call the doc object's Save() method to save the current XmlDocument to the file system.

Opening a File

Opening a file also uses a familiar method of XmlDocument. I used another File dialog to let the user choose a file to open. Next, I used the Load() method of doc to load the file into memory. I then called the resetQuiz() method to prepare the quiz for editing.

```
private void menuOpen_Click(object sender,
        System.EventArgs e) {
  if (opener.ShowDialog() != DialogResult.Cancel){
    doc.Load(opener.FileName);
    theTest = doc.ChildNodes[1];
    resetQuiz();
  } // end if
} // end mnuOpen
```

Adding a Question

To add a question to the end of the quiz, I used the algorithm described in the XML Creator program described earlier in this chapter:

```
private void mnuAddQuestion_Click(object sender,
        System.EventArgs e) {
  numQuestions++;
  qNum = numQuestions -1;

  //create the new Node
  XmlNode newProblem = theTest.ChildNodes[0].Clone();
  theTest.AppendChild(newProblem);

  showQuestion(qNum);

  //clear the screen
  txtQuestion.Text = "";
  txtA.Text = "";
  txtB.Text = "";
  txtC.Text = "";
  txtD.Text = "";
  optA.Checked = false;
  optB.Checked = false;
  optC.Checked = false;
  optD.Checked = false;

} // end mnuAdd
```

The mnuAddQuestion_Click() method begins by making a clone of the first problem and then appending the clone to the end of the test. I incremented numQuestions to indicate that the total number of questions has changed, and I set qNum to indicate the last question, which is the new node that has just been created.

I didn't bother to change the values of the new node (they will still be exactly the same as the values of the first problem node) because it isn't necessary to do so. I will clear the screen, and when the user moves off this question, the values on the screen will be copied over to the new node with the updateQuestion() method.

Creating a New Test

Creating a new test uses another algorithm developed in the XML Creator program from earlier in the chapter. I modified the code to reflect the structure of the quiz markup language I had devised:

```
private void mnuNewTest_Click(object sender,
        System.EventArgs e) {
  doc = new XmlDocument();
  theTest = doc.CreateNode(XmlNodeType.Element, "test", null);

  //create the first line:
  //<?xml version="1.0" encoding="utf-8"?>

  XmlNode header = doc.CreateXmlDeclaration("1.0", "utf-8", null);

  XmlNode theProblem = doc.CreateElement("problem");
  XmlNode theQuestion = doc.CreateElement("question");
  XmlNode theAnswerA = doc.CreateElement("answerA");
  XmlNode theAnswerB = doc.CreateElement("answerB");
  XmlNode theAnswerC = doc.CreateElement("answerC");
  XmlNode theAnswerD = doc.CreateElement("answerD");
  XmlNode theCorrect = doc.CreateElement("correct");

  //construct the new node structure
  doc.AppendChild(header);
  doc.AppendChild(theTest);
  theTest.AppendChild(theProblem);
  theProblem.AppendChild(theQuestion);
  theProblem.AppendChild(theAnswerA);
  theProblem.AppendChild(theAnswerB);
  theProblem.AppendChild(theAnswerC);
  theProblem.AppendChild(theAnswerD);
  theProblem.AppendChild(theCorrect);

  //populate values of nodes
  theQuestion.InnerText = "question";
  theAnswerA.InnerText = "a";
  theAnswerB.InnerText = "b";
  theAnswerC.InnerText = "c";
  theAnswerD.InnerText = "d";
  theCorrect.InnerText = "X";

  qNum = 0;
  showQuestion(0);

} // end mnuNewTest
```

First, I created a new XmlDocument and XmlNode to hold the document and the test, respectively. I then created the XML header with a call to the XmlDocument's CreateXmlDeclaration() method.

I then created a series of XML elements to hold the problem and its components. Next, I used the AppendChild() method of the various nodes to build the test and the first problem. Finally, I added some default text to each of the nodes. I then reset qNum to 0 and showed the initial question.

Summary

In this chapter you learned how C# can be used to create and explore XML documents. You have discovered the internal structure of XML documents, learned how to create an XML document in the IDE, and learned how to write a program that can navigate through any XML document. You have also learned how you can generate an XML document from scratch or add an element to an existing document.

Challenges

- Modify the Adventure Kit game in Chapter 9 so that it uses XML data instead of object serialization to store the adventures.
- Create an address book or another simple database, using XML technology.
- Investigate existing XML technologies such as SMIL or SVG, and build a simple subset.
- Create a new XML language to describe a type of data you frequently work with.
- Use the XML skills you have learned to read an HTML or XML document and display selected information (perhaps it should switch to a larger font whenever your name is encountered in the document).

Chapter 11: Databases and ADO.NET: The Spy Database

Overview

As you have grown more experienced in C#, you have seen the importance of data for creating interesting programs. Often you will find that you have a large amount of data that has already been generated in some sort of database management package, such as Access, Oracle, or SQL Server. C# provides a set of classes for working with existing databases. It also provides very handy tools for creating a database. In this chapter you will get exposure to the complex and powerful world of data manipulation and learn to

- Build a simple database using the built−in data management tools
- Create a form that uses data connections, data adapters, and data sets to access data
- Use basic data normalization principles to design multi−table databases
- Build relationships and views for managing complex databases
- Connect to various types of databases and raw XML data as a data source
- Update data on the fly

A Note about the CD−ROM

This chapter focuses on techniques for building databases on your own server. The examples in this chapter outline how I designed a data system on my own server, but the programs on the CD−ROM will not run unless you build the databases on your machine. In the event that you don't want to build the database, I have included an Access database (spy.mdb) on the CD−ROM that you can connect to using the techniques outlined in the section, "Working with Other Databases."

Introducing the SpyMaster Program

To illustrate the database generation features, you will build a simple database used to manage your international network of spies (you *do* have an international spy network, don't you?).

The Spy Master program main screen is illustrated in Figure 11.1.

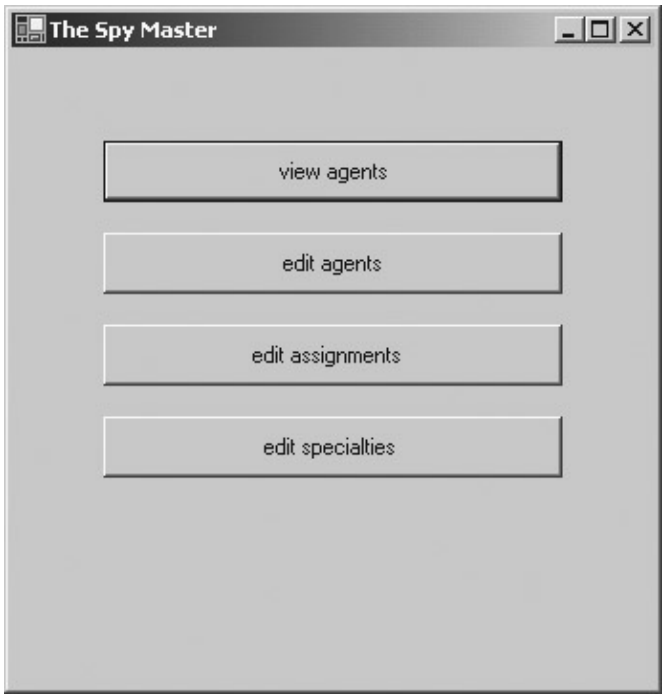


Figure 11.1: From this main screen you can achieve world dominance (or protect it from evil).
The main screen features familiar buttons that take you to other forms. To view your agents, click on the (what else?) View Agents button. You will see the form shown in Figure 11.2.

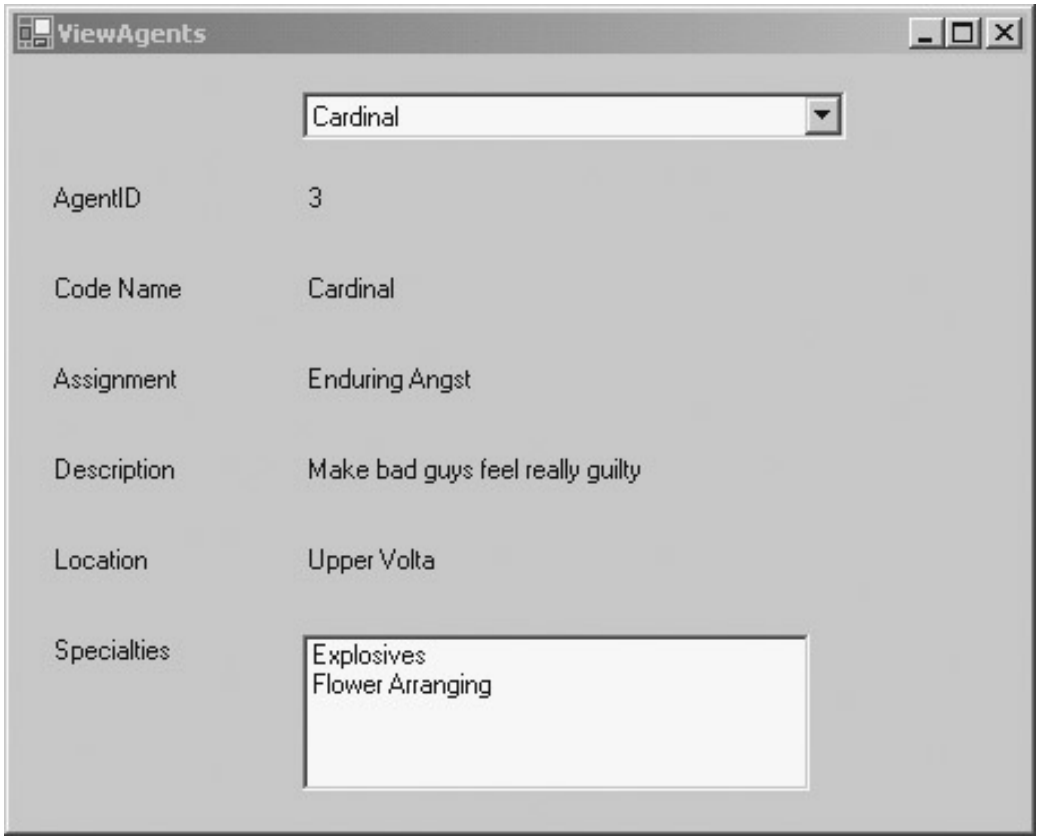


Figure 11.2: All the information regarding a spy is available on this form.
You can choose any agent in your roster from the drop–down list at the top of the form.

Because you are the spy *master* you also can change spy information at will, letting slip the dogs of international intrigue, as shown in Figure 11.3.

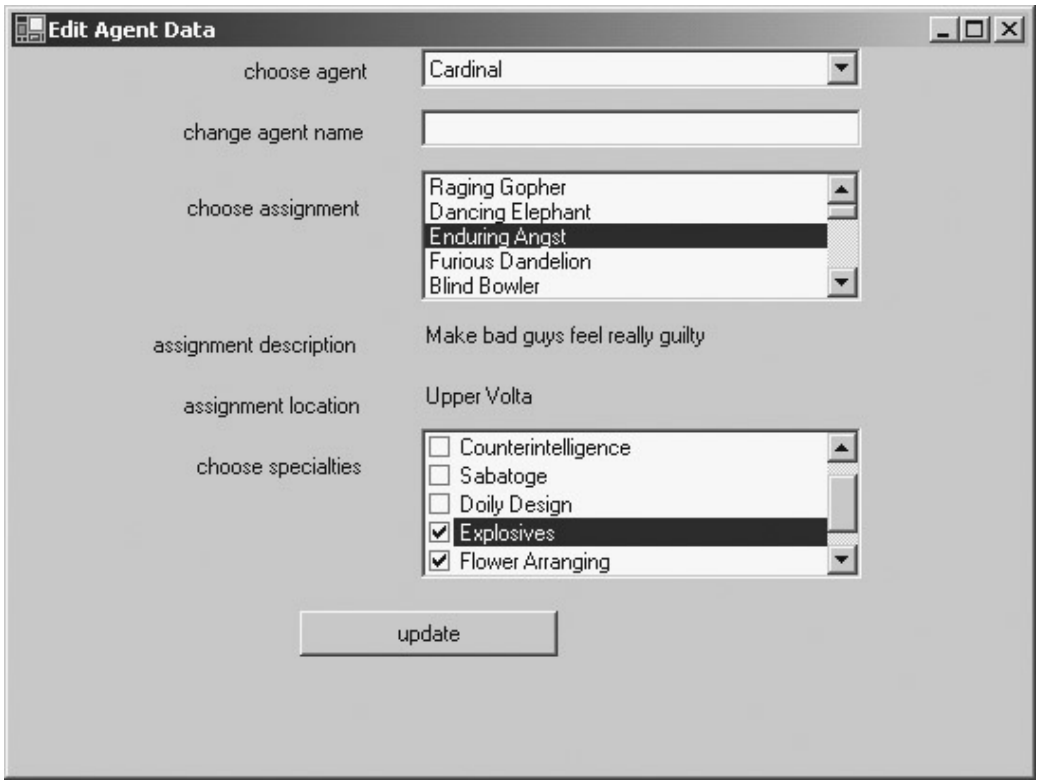


Figure 11.3: The edit agents screen is similar to the view agents screen, except that now you can change some of the data.

You can change a lot from the edit agent screen, but you can't change *everything* from this screen, because designing daring missions of danger and creating specialties are different than manipulating your pawns on the grand chess table (evil laughter). There is a separate form for editing missions, shown in Figure 11.4.

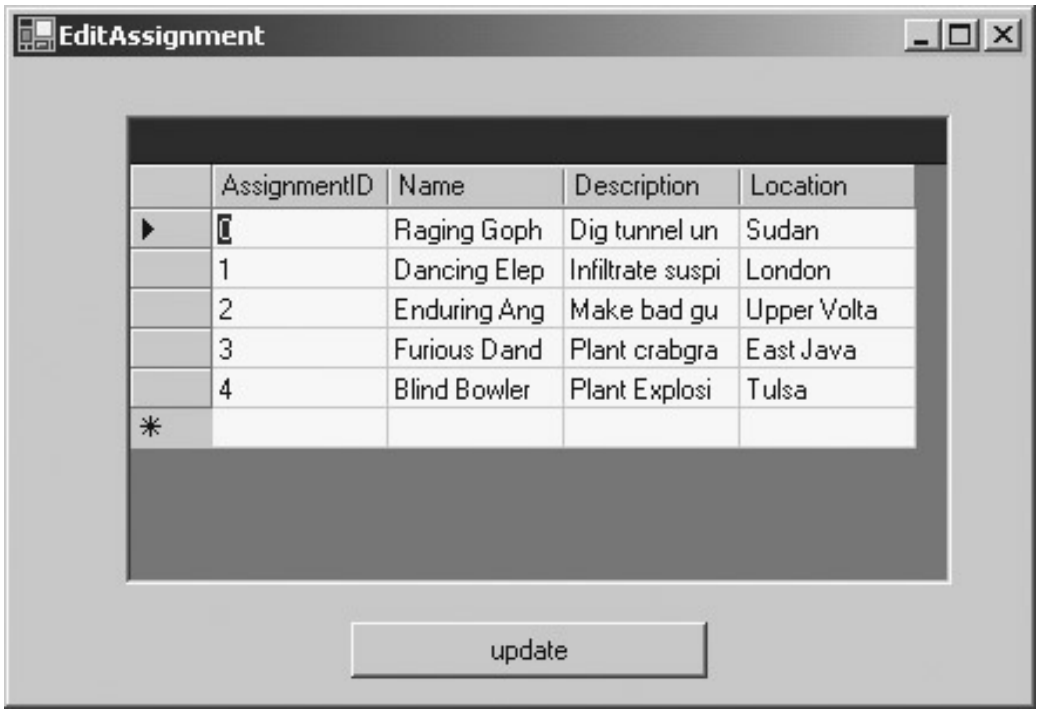


Figure 11.4: Here the user can add and modify assignments in a grid.

Finally, you can add to the specialties your agents can pursue.

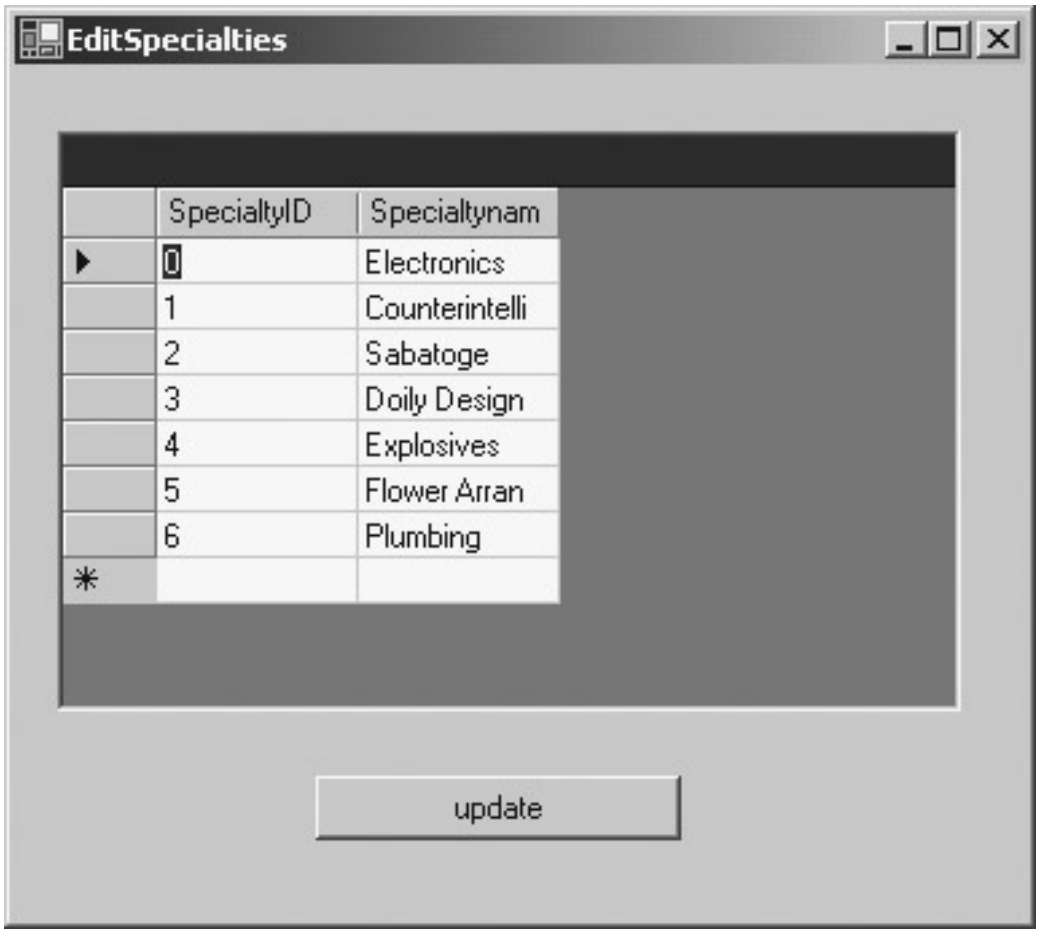


Figure 11.5: The user can add and modify specialties. You never know when explosives and doilies will be needed on a mission.

Creating a Simple Database

It pays to start small. You'll begin with a more modest version of the spy database. Although you can write programs to generate a database, it's often easier to use a database management system to create the original database. Microsoft includes a very powerful subset of the SQL Server database program as part of the visual studio editor. If you have any experience with Microsoft access (or pretty much any other relational database package) you'll find the tools for data design to be pretty standard.

Hint Not all installations of Visual Studio.NET include the SQL Server extensions. If you do not have this functionality, you can follow the same steps outlined in this chapter in Microsoft Access or nearly any other modern database program, and then you can use the steps outlined in the "Working with Other Databases" section later in this chapter to connect to the database. Nearly every modern data package will have some variation of the tools I describe in this section, although the exact techniques for accessing these tools varies.

Accessing the Data Server

The toolbar tab on the left side of the IDE has been hiding some important secrets from you. The Server Explorer tab at the bottom–left of the toolbar brings to focus an entirely new set of tools you might never have seen before. Being the international spymaster you are, you can master these new tools quickly. The server explorer enables you to examine many kinds of tools available on the server or servers your computer is currently connected to. The term *server* is used to describe a

computer that provides information, and also to the software on the computer that distributes information. You may be familiar with network servers, which provide a foundation for a local area network, or Web servers, which is where Web pages are stored and accessed. There are many other types of servers available as well, including data servers, which are used to manage databases. When you installed Visual Studio, a simplified version of SQL Server, Microsoft's powerful data server, was also installed on your machine. You can use this relational database package within the Visual Studio IDE to develop your own databases, and integrate them into your programs.

Creating a New Database

If you look at the servers tag, you'll see a list of servers you are connected to. As a default, only your own machine name is listed.

> **Hint** When I installed Windows 2000 on my work machine, I went with the default machine name that Microsoft suggested. That was a really bad idea, because I can hardly even type that monster, let alone remember it. In the following code and screen shots, replace andy–mpecr6vc86 with your own machine's name. If you haven't yet installed Windows 2000 or XP, make sure you choose a machine name that's easier to remember than mine.

My server is listed, so I clicked on the corresponding plus sign to see the various parts of the server that are available. SQL servers are near the bottom of the list. SQL (Structured Query Language) is the name of a powerful database manipulation language. You get a brief introduction to it in this chapter. If you expand the SQL Servers item, you see your machine name indicated again. Finally, you see a list of the actual databases registered to your machine. Even if you haven't attached any databases yet, there will still be a few placed there as prototypes. Right–click on your machine name under SQL Servers and you are given an opportunity to create a new database. Do so, and name it "SimpleSpy." When you are finished, the IDE adds the SimpleSpy database and several related elements to your server list. It should look like Figure 11.6.

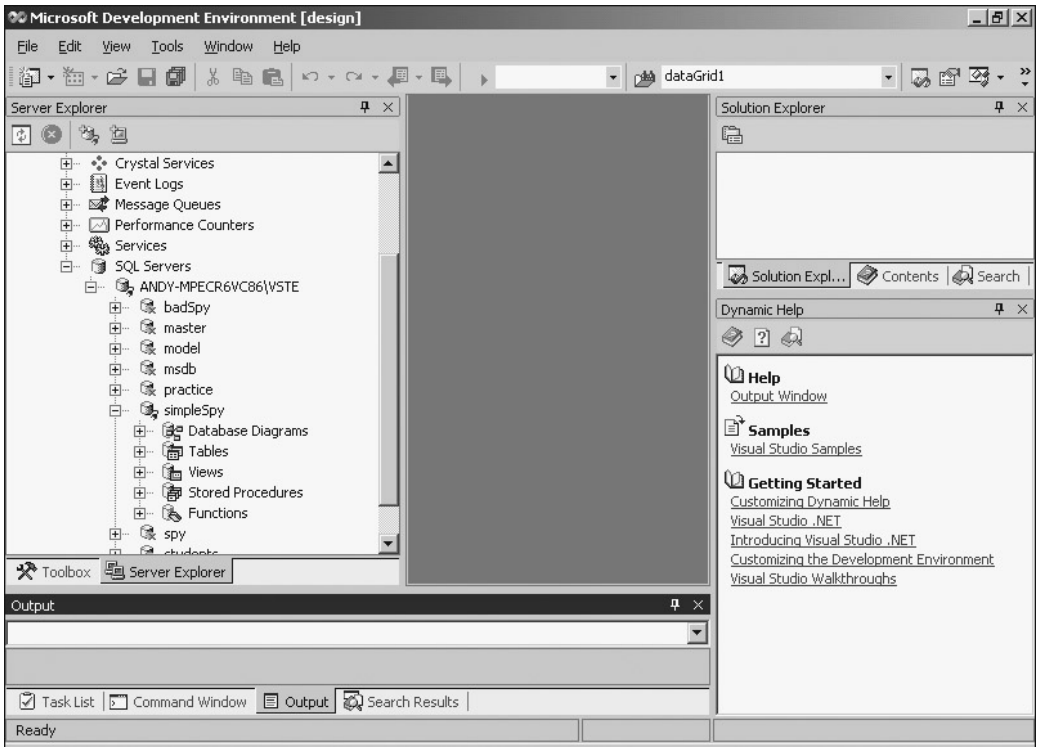


Figure 11.6: The hierarchy tree on the left–hand side shows my machine with my machine listed as an SQL Server. The SimpleSpy database is listed as an element of my SQL Server.

Adding a Table

Databases use a construct called a *table* as the core entity for storing data. A table holds information about a specific element. Each table consists of a number of *fields*, which each have a value and a type. To make a new table for the simple spy database, right–click on Tables under simpleSpy in the server explorer tab, and choose Add New Table. The main panel of your editor changes so it looks like Figure 11.7.

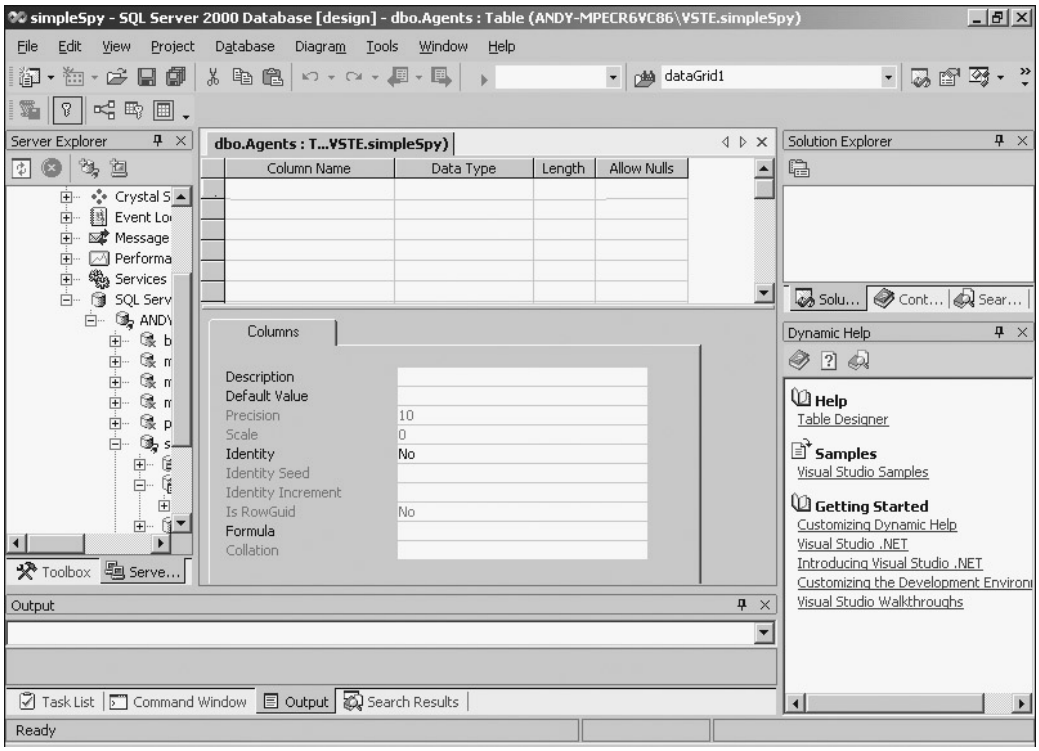


Figure 11.7: A table editor reminiscent of Microsoft Access pops up.

If you have any experience with Microsoft Access (Microsoft's very popular entry–level database) the table design screen probably looks very familiar to you. The table designer prompts you for field names and types. The Agents table is designed to describe a list of secret agents. Each agent (at least in this current simple form) has three characteristics. Figure 11.8 illustrates the first stage of table design for the Agents table.

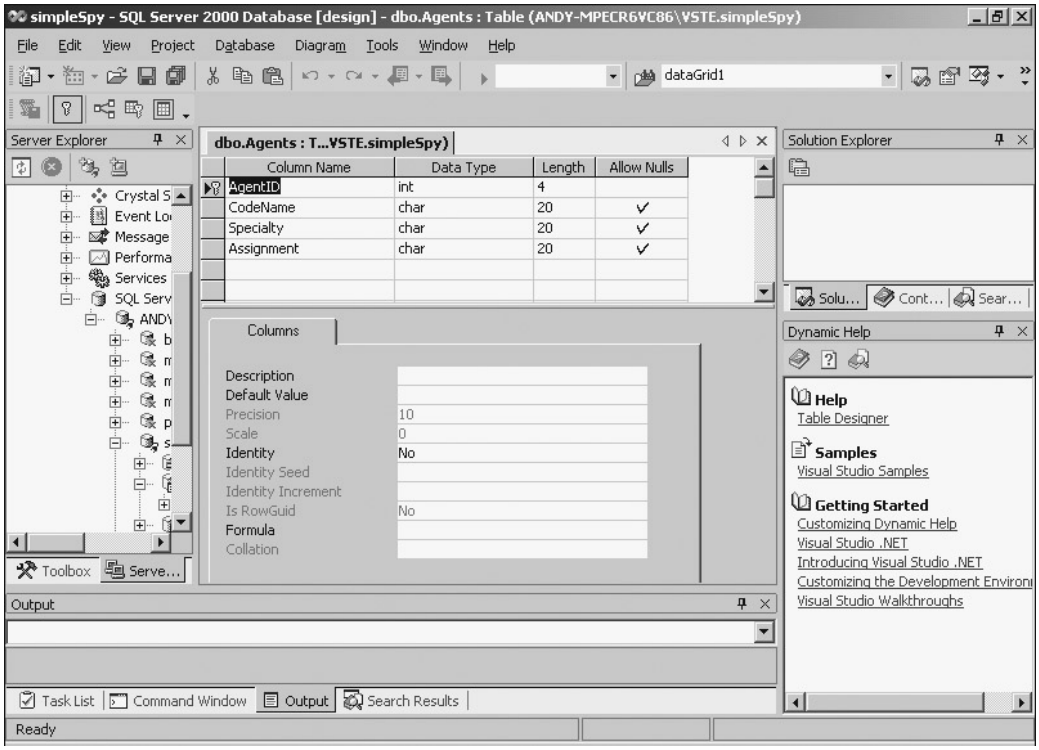


Figure 11.8: Each agent has a code name, an assignment, and a specialty field.

For each field, you must determine a field type and length. The field types are much like the variable types you are already used to, but there are a few differences. Fortunately, all the legal field types are available in the drop–down list. Note that text values are listed as char rather than string. This is because data definition has a different tradition and nomenclature than programming. If a field is defined as a text field, you are required to list the length of the field in characters. The software determines where one record ends and another begins by counting the number of bytes in each record. Every record in a table must be the same length. If you define a field to be 10 characters long, any values longer than ten characters will be truncated, and trailing spaces will be added to any values shorter than ten characters. Field names should be single words, like variable names. (Many database packages allow multiple word field names, but they usually make working with the data much more complex.)

Trap The Simple Spy database has some serious problems in its design, as you'll see. I'll show you how to improve the database throughout the chapter, but first I want to explore attaching this simplistic database to a form and displaying it in a C# program.

Creating a Primary Key

Notice that the first field in the Agents table is called SpyID. This field is meant to provide an index for the Agents table. It is much easier to work with large and complicated data sets if each table has a special field identified as its primary key. You have already encountered primary keys many times. Every time you make a call to your insurance company, the bank, your credit card company, or any other bureaucracy, somebody asks you for an account number. They are rarely concerned about your name as long as they have that account number. An account number is usually a primary key to the data about you in their database. The primary key is used to find all the other information about you in the database. It would theoretically be possible to use your name as a primary key, but this doesn't work very well, because of spelling and capitalization problems. There are also people with duplicate names. Account numbers—and primary keys—are designed to avoid these problems. A primary key should be unique—the key should be different for each record in the database. Primary keys should also be universal—each record must have one. Finally, primary keys should be

non–null. (This seems obvious, but in many databases, you have to specify that a field is non–null explicitly, so it's worth mentioning). The best keys are integers, because you don't have to worry about spelling or capitalization.

In most of my tables, I create a primary key as the first field in the table, and I usually end the field name with ID. To specify in Visual Studio that a field should be used as a primary key, select the entire field by clicking on the triangle to the left of the field, and click the key icon of the database toolbar at the top of the screen. A small key icon appears beside the field in the definition table, as shown in Figure 11.9. Notice that when you set a field as the primary key, the allow nulls check box is automatically deselected.

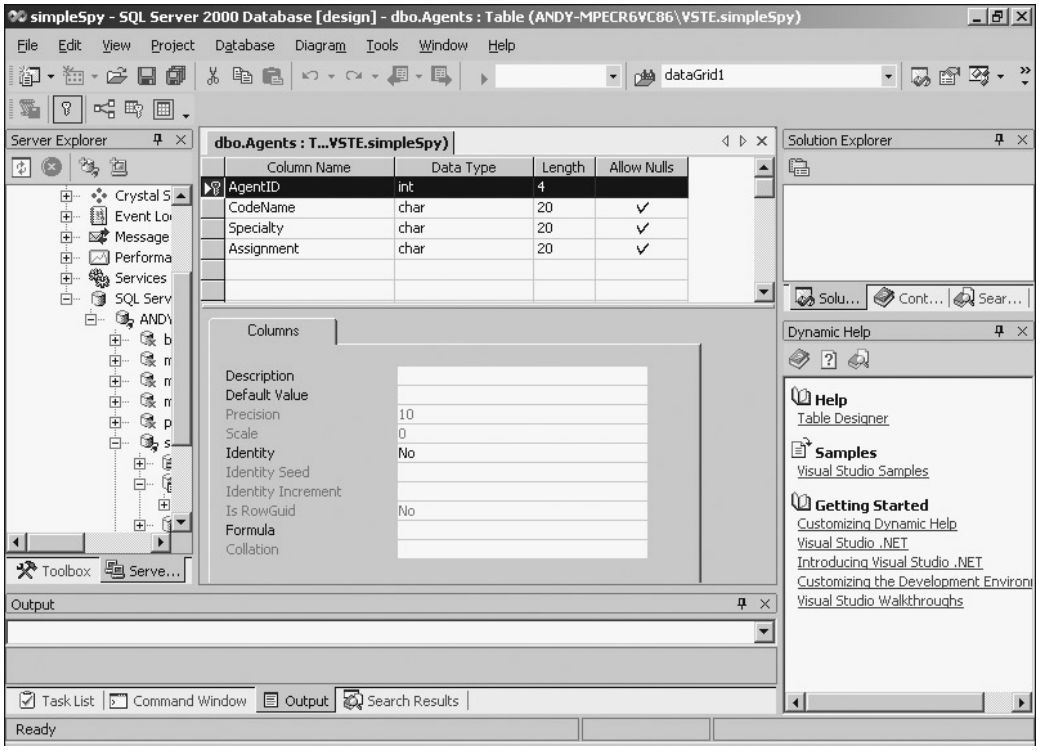


Figure 11.9: The AgentID field is now the primary key of the Agents table.

Hint It's very smart to add a primary key to every table, even though you might not feel it's necessary. When you start to build relational databases later in this chapter, you'll see that primary keys are critical to building well–formed data structures. If in doubt, add an integer field, call it an ID field, and designate it as a primary key. It never hurts to have one.

To finalize your design of the table, close the window containing the table definition. A dialog pops up asking you the name of the table. Call the new table Agents.

Adding Agents to the Table

After you have created the Agents table, look again at the server explorer. You see that the Agents table is registered to your server, as illustrated in Figure 11.10.

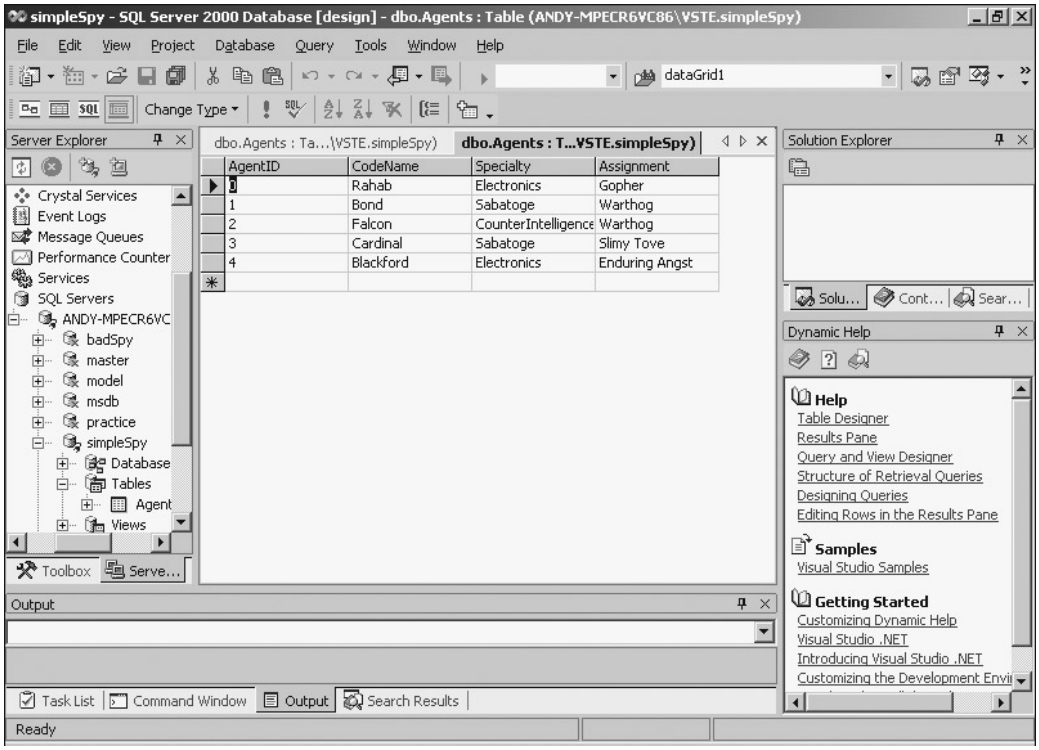


Figure 11.10: Agents table after I've entered a few of my agents. (This page will self–destruct in 5 seconds…)

Keeping Track of Your Data Structure

The way the server explorer displays the structure of your database in the IDE is a welcome addition. You used to be able to tell which programmers were working on databases because they always had paper diagrams of their data structure tacked to all the walls in their cubicle. (Many pros still do this.) In fact, there are stories of data programmers taking only their data diagrams as they left a burning building. The problem with paper diagrams is they have to be replaced every time the data structure is changed. The server explorer gives you an easy way to see exactly what tables are in your database, and what fields belong to each table. This seems like trivial information to keep track of right now, but the complexity of databases seems to grow very quickly.

Accessing the Data in a Program

Now the spy data is available to my server, but it isn't very interesting there. I want to be able to use the data in programs to save the world for democracy, apple pie, and quarter video arcades. It shouldn't surprise you that the IDE makes it easy to integrate a database with a program. Once you've created a database on your server, you can easily access it from any programs written on that same machine (later on I'll show you how to get to other databases, too). To illustrate this, create a project (or simply go to a form if you're already in a project). C# has a special control designed explicitly for working with databases. It's called the *Data Grid* control. To use it, display a form in the visual designer, and go back to the traditional toolbox (with all the form components you're used to placing on the screen). The data grid can be dropped on the form like any other control. It is not very interesting unless it is connected with some form of database. To begin creating the data connection, drag the Agents table from the server connection onto your form. Figure 11.11 illustrates the visual designer after I have added a data grid control and dragged the Agents table to the form.

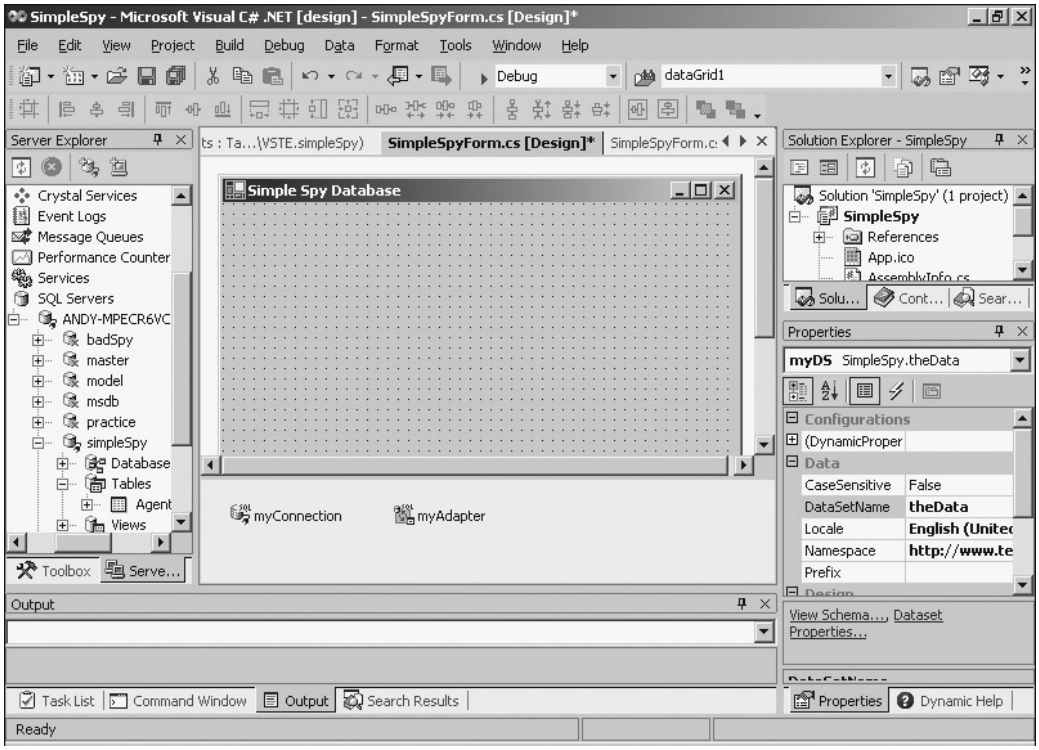


Figure 11.11: When you drag a table to your form, two new objects are created in the non–visible segment of your form.

Dragging a table to the form causes the IDE to create two objects. A SqlConnection object describes the connection between your program and the database. There are other kinds of connection objects, but they all do basically the same thing. A connection object encapsulates the actual connection. If you want to connect to a database on a remote system, you can set up the connection object to point to the remote machine, handle any logins, and start the connection.

The most critical property of the SqlConnection is the ConnectionString property. Take a look at it in the designer; it's a big ugly string variable designed to specify how to connect to the database. It's usually easiest to let the designer automatically generate an SqlConnection object, (and the complicated ConnectionnString property) and then modify it as needed.

The other object that is automatically created when you drag a table to the screen is an instance of SqlDataAdapter. The adapter classes are a new feature of the .NET data access scheme. In a nutshell, a data adapter automatically generates a local copy of a database on the client's computer, and manages the relationship between the local copy of the database and the original database. This separation is especially useful when you are working on databases across Internet connections, but it works just as well when the database is local. You'll generally work with the SqlDataAdapter's Fill() method when you want to request data from the main database, and the Update() method when you want to update the main database from your local version.

Creating a DataSet Object

Although the SqlConnection and SqlDataAdapter are handy, you need one more data class to work directly with data in your forms. This important class is called the DataSet class. Each database uses a custom extension of the DataSet class to provide access to the database. Fortunately, .NET makes it easy to generate an appropriate DataSet from a data adapter. To create the new DataSet, look at the properties box of the DataAdapter class. At the bottom of the properties box, you see links for some special commands. Choose Preview Data, and you see the dialog box featured in Figure 11.12.

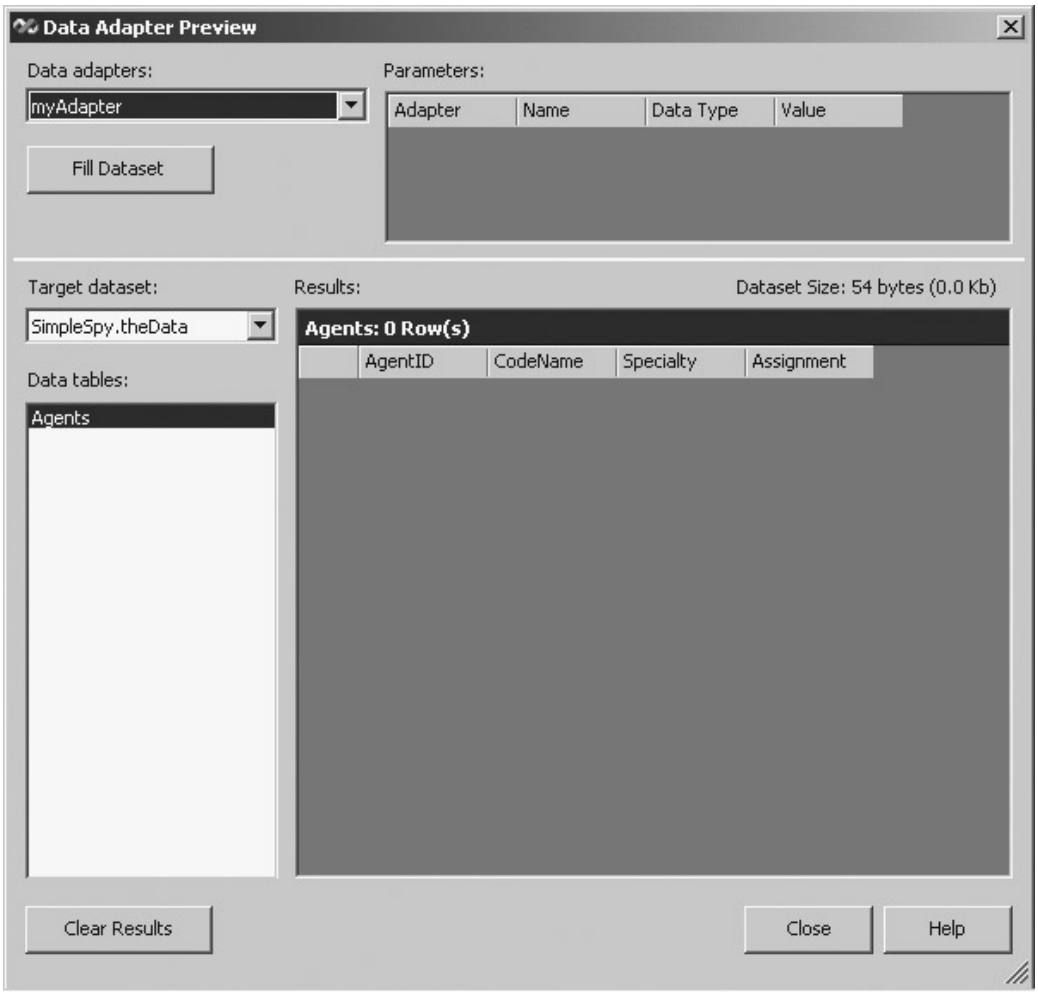


Figure 11.12: The Preview Data dialog illustrates how the data looks to the program.

How Do Data Connections, Data Adapters, and Data Sets Fit Together?

All this new (and similar) terminology can be baffling. It actually makes sense if you understand how Microsoft's underlying data model (called ADO.NET) works. Essentially, your programs are considered completely separated from your actual database, even if they are on the same machine. In the .NET model, databases and programs have the same kind of long–distance relationship that Web browsers have with Web servers. In the Web, a browser requests a page, and that page is loaded onto the local machine. The ADO.NET data model works in a very similar way. Think of the DataConnection object as being your placeholder for the actual (remote) database. The DataSet object holds a local copy of the data. You can manipulate the DataSet object all you want, but it doesn't directly affect the original database. The DataAdapter class is the conduit between the (remote) DataConnection and the (local) DataSet. The DataAdapter reminds me of the pneumatic tubes sometimes used at bank drive–through windows to send checks and pens from the bank to the cars. It is a communication medium. You use methods of the DataAdapter to fill up the DataSet with data from the DataConnection, and you use other methods of DataAdapter to update the original database from the local DataSet. If you're still confused, read on. You'll get it after a little more exposure.

The Preview Dialog creates a temporary DataSet based on a particular data adapter. Because the

current program has only one data adapter available, clicking on the Fill DataSet button creates the temporary data set based on the current data adapter, and displays the results in the dialog, as illustrated in Figure 11.13.

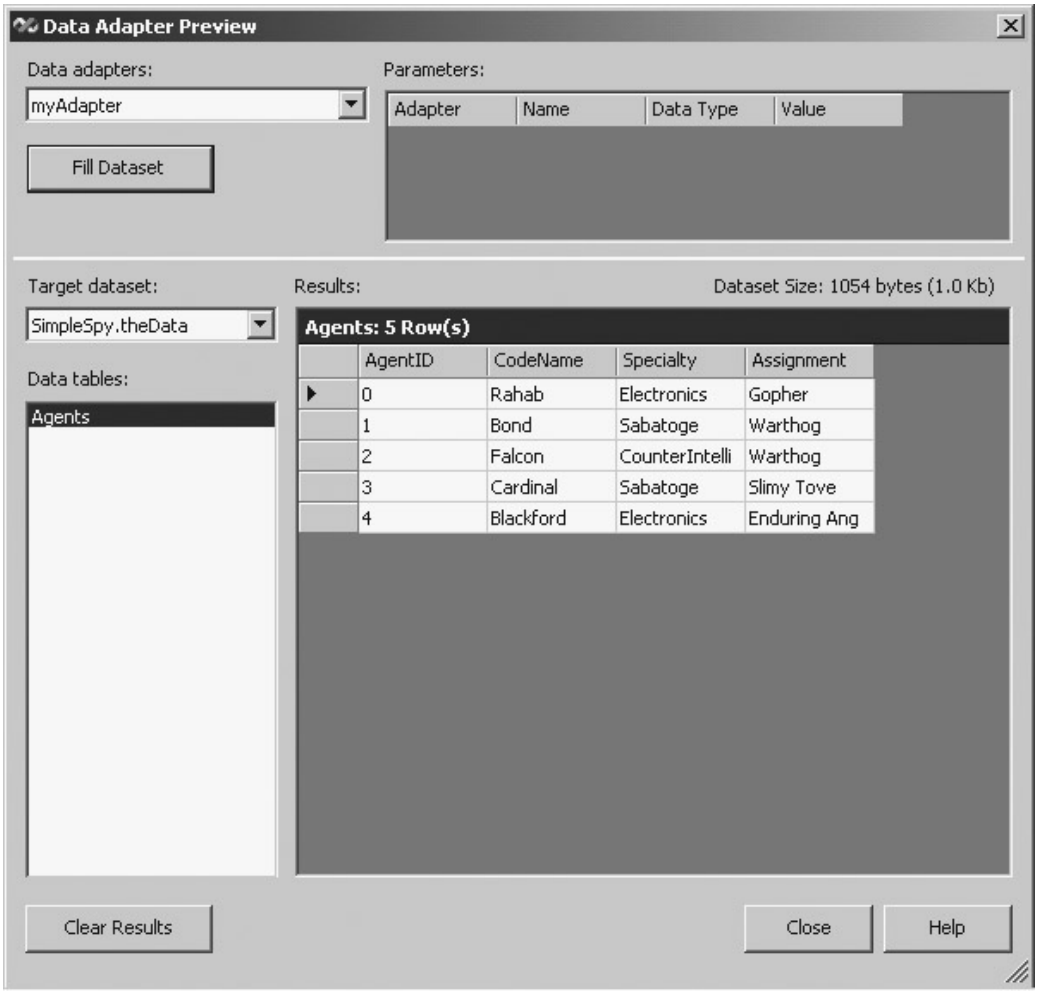


Figure 11.13: After pressing the Fill DataSet button, the data set appears on the dialog.

Once you have confirmed that the database acts as you expect, close the Preview Data dialog and look again at the bottom of the Properties Window for the Generate DataSet command. Click on this link to get the Generate DataSet dialog, shown in Figure 11.14.

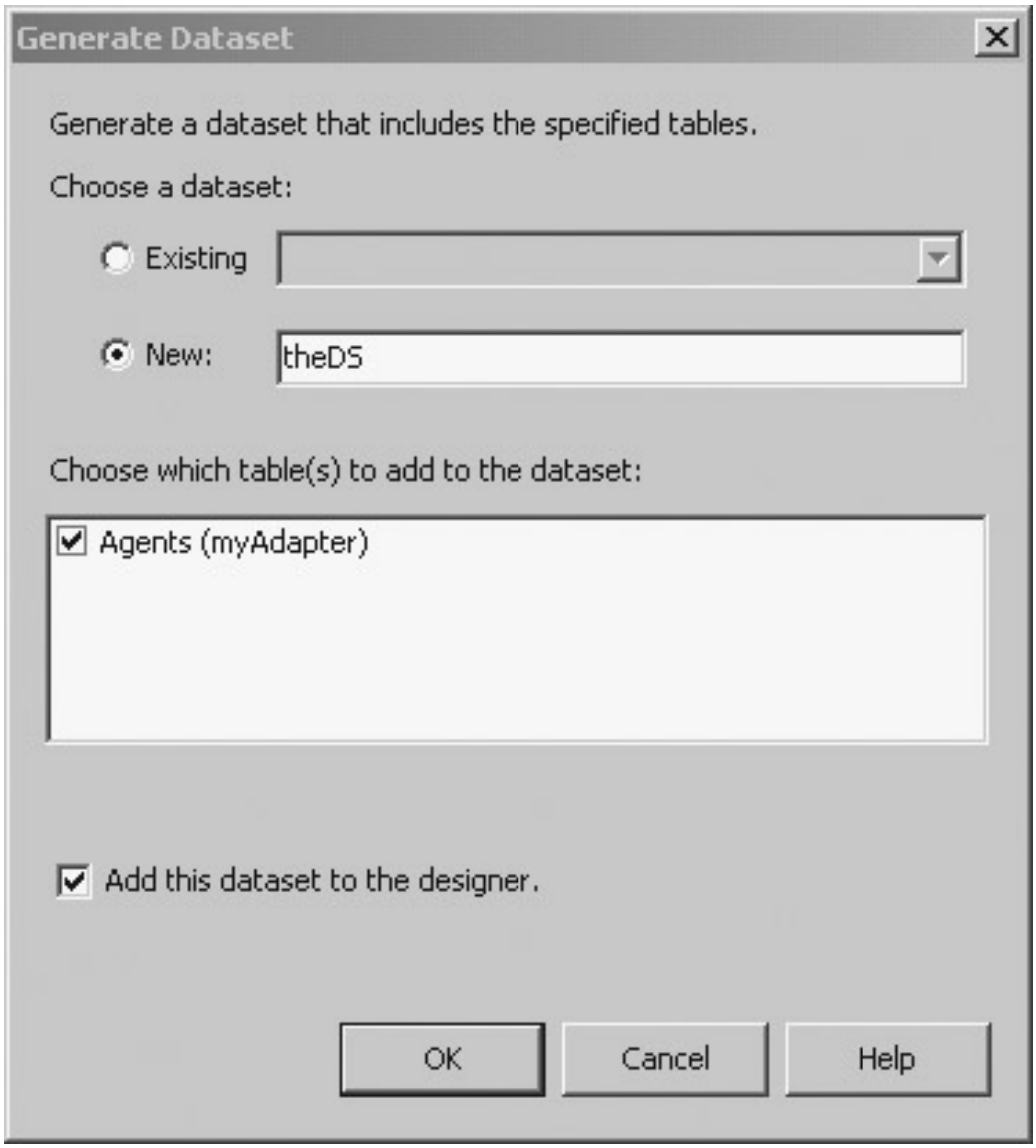


Figure 11.14: The Generate DataSet Dialog prompts you to name your new DataSet.
To create a new DataSet, choose the New radio button, then type in the name of your new data set class.

Trap When you created the connection and adapter classes, you were making *instances* of existing classes. The data set is different, because you are creating an entirely new class that extends the base DataSet class. The Generate DataSet dialog creates both the new data set class and an instance of that class. That's why when you leave the dialog, the data set object at the bottom of your form has a one at the end (the default name for any instance of a class is the class name followed by an integer). If you do not rename the data set, it is called DataSet1, and its first instance is called DataSet11. I was quite confused by this until I followed my own advice from way back in Chapter 1, "Basic Input and Output: A Mini Adventure." When you make new things, you should rename them. As you can see in Figure 11.14. I called my data set theData.

Once you leave the Generate DataSet dialog, you are returned to your form and a new instance of your data set appears at the bottom of the form near the connection and adapter objects. Because I named my data set theData, the default name for the new data set is theData1. I renamed the instance myDS, to help me remember this is my custom data set.

Connecting the Data Set to the Grid

The data grid control is designed to be *bound* to a data set. This allows an automatic connection between the grid and the data set. To bind myDS to the data grid, set the DataSource property of the grid to myDS, and the DataMember property to Agents. Both properties generate drop–down list boxes if the data source has been connected properly. Figure 11.15 illustrates setting up a data grid.

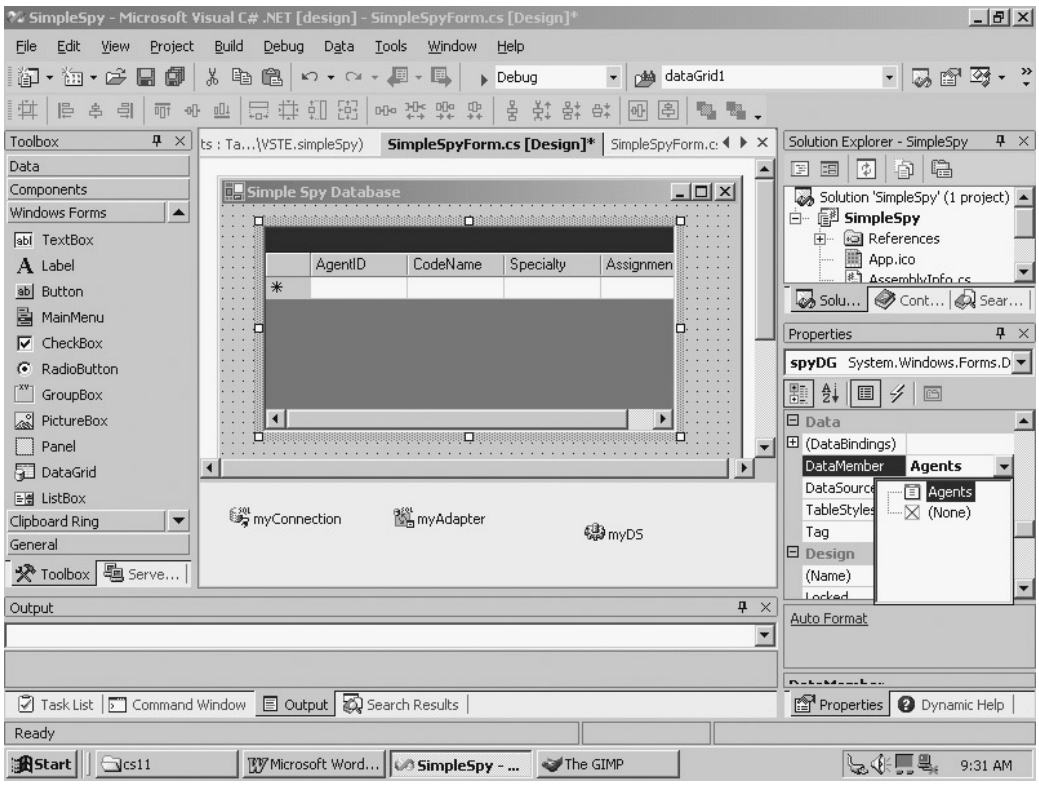


Figure 11.15: Setting the data DataMember property only works after you've set the DataSource property.

The DataSource property determines what DataSet object the grid will be connected to. DataSets can (and will, later in this chapter) have more than one table, so the DataMembers property is used to determine which table (or other entity) is connected to the grid.

Filling the DataSet from the Adapter

The database is almost ready. Figure 11.16 shows the data grid with its DataSource and DataMember properties set appropriately.

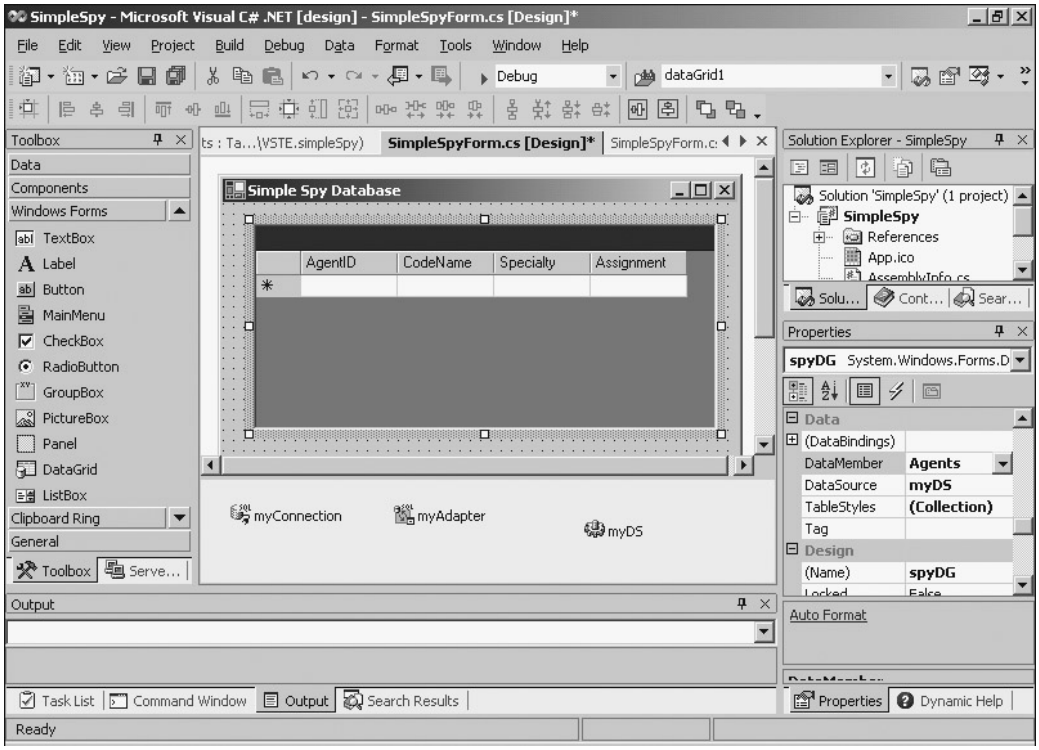


Figure 11.16: The program is almost ready, but the actual data has not yet been passed to the data set from the original database.

The only thing left to do is fill up the data set. Recall that the data set is simply a local copy of the database. The DataAdapter handles the connection between the original database and the copy stored in the data set. I added one line of code to the form's load method to fill up the data set.

```
private void SimpleSpyForm_Load(object sender,
        System.EventArgs e) {
  myAdapter.Fill(myDS);
} // end form load
```

The adapter's Fill() method requires a data set as a parameter. It goes to the original database and grabs the relevant information (determined by the properties of the adapter class) to populate the data set.

You can now run the Simple Spy program to get the results shown in Figure 11.17.

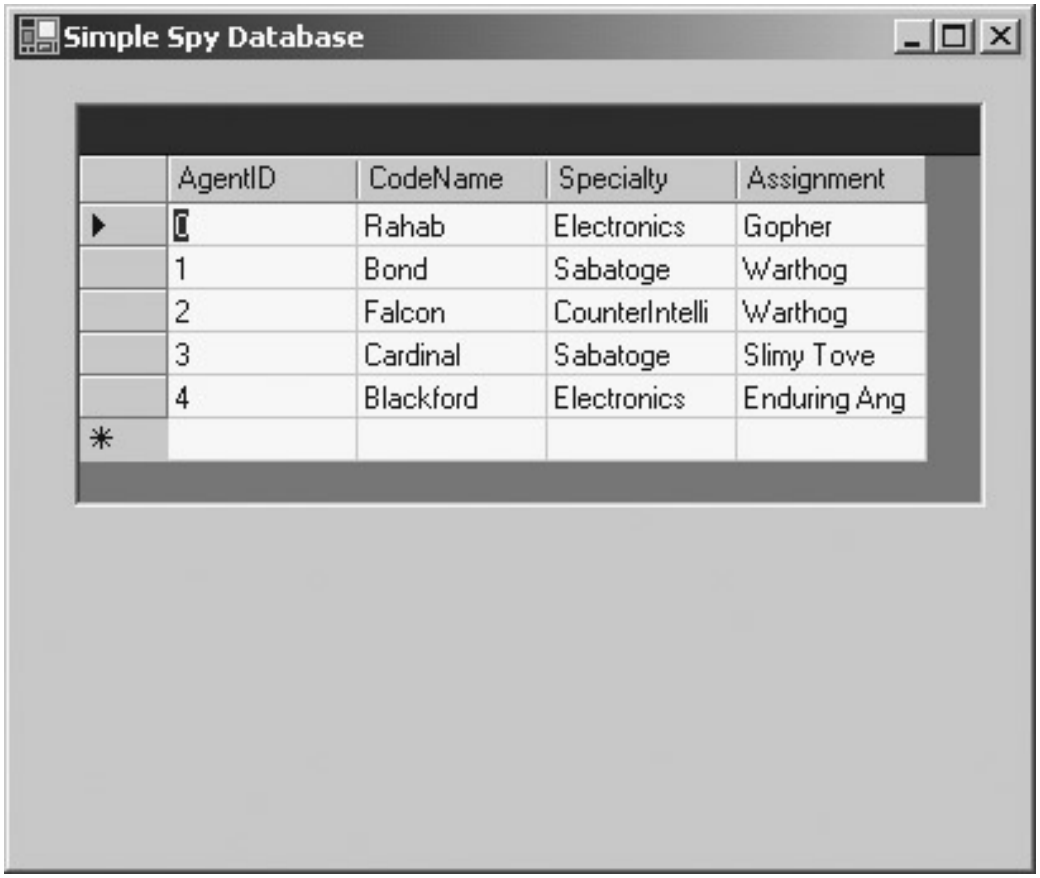


Figure 11.17: Although there was a lot of mouse jockeying involved, the database can be displayed on a form with only one line of code.

Using Queries to Modify Data Results

The interesting thing about databases is that they usually have *too much* information to be immediately useful. The main thing a user needs to do with a database is screen out information to find exactly what he or she is looking for. Most data management systems use a query to provide this information. The query demo program shows you how queries work in Visual Studio.

Limiting Data with the SELECT Statement

Figure 11.18 demonstrates a new version of the spy statement that enables the user to limit the database results.

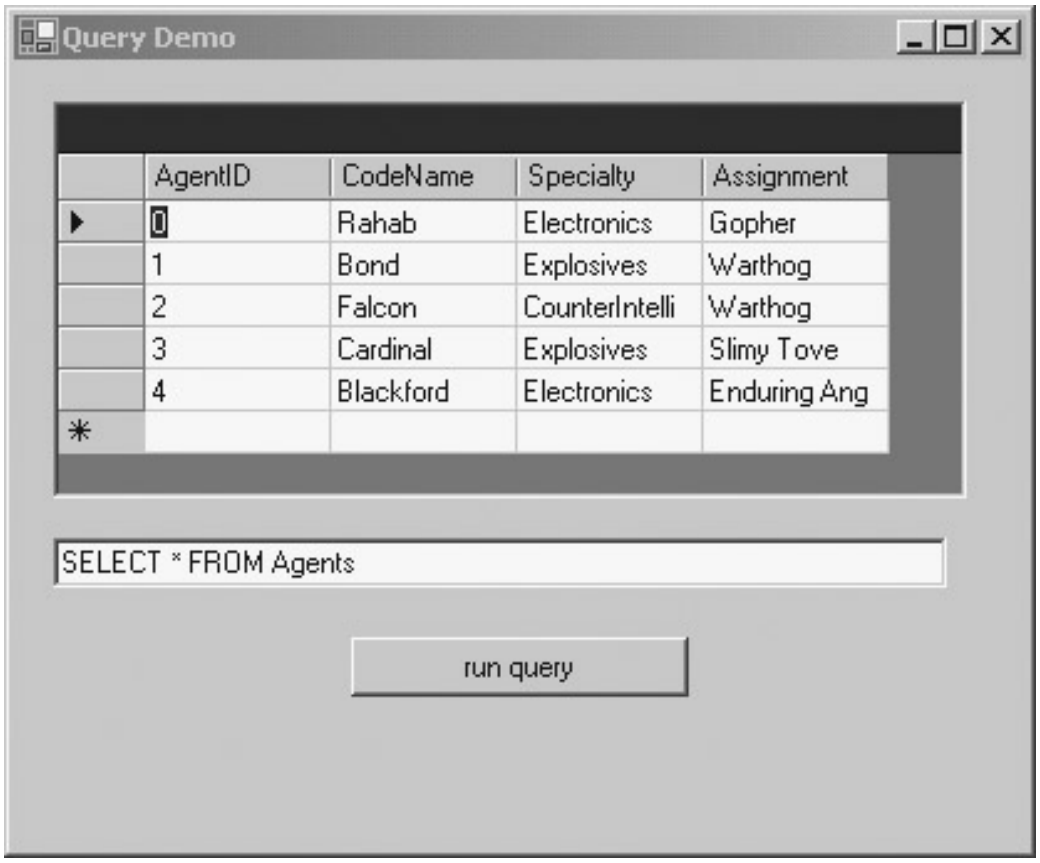


Figure 11.18: The default version shows the entire database of spies.

The text box below the grid enables the user to type in database manipulation commands in a special language called *Structured Query Language* (SQL). SQL features a number of commands for data manipulation, but the SELECT command shown here is the most basic and most commonly used command. The asterisk (*) means "all fields", so the command SELECT * from Agents can be read "Display all the fields of the Agents table." As you can see, this is exactly what the program is doing at the moment. You might want a quick list of your spy's codenames, without any other information. To limit the grid so it only displays codenames, you can type the following command into the text box and then click the Execute Query button:

```
SELECT CodeName FROM Agents
```

Figure 11.19 shows what happens when this statement is executed.

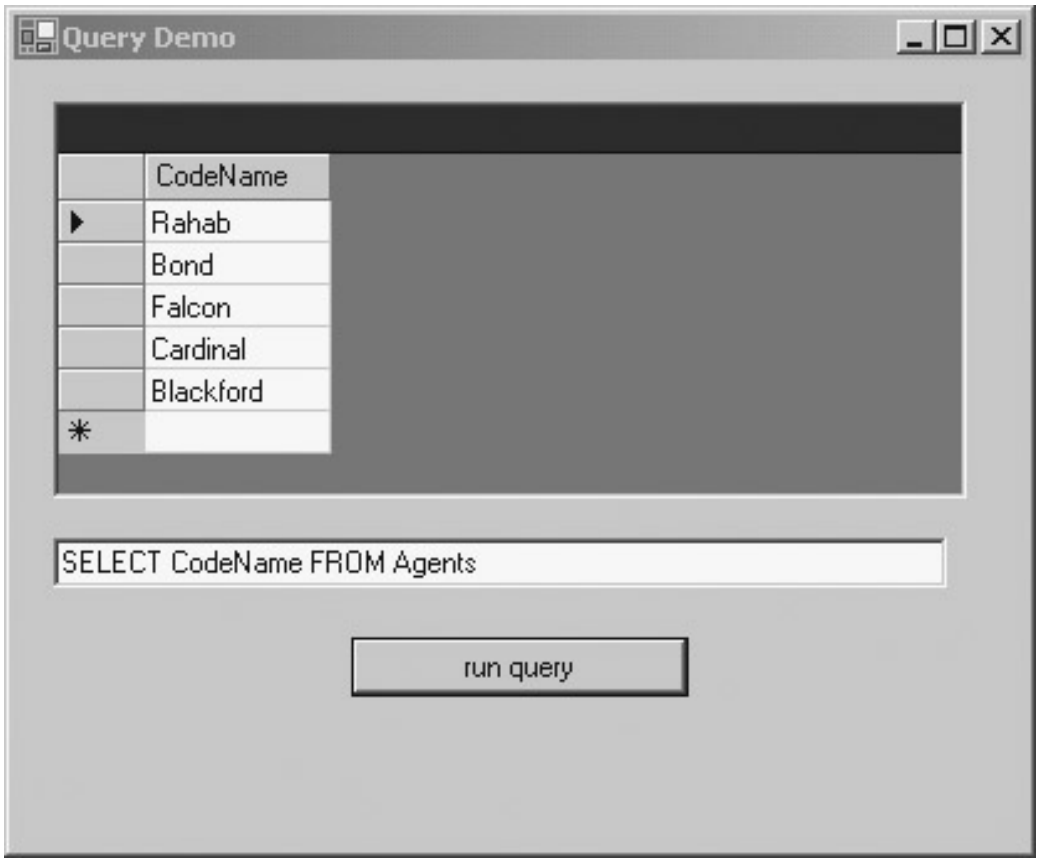


Figure 11.19: Now the program only shows the spy codenames.

The keyword SELECT indicates that you are going to get a subset of the original table. CodeName is the name of a field you wish to view. The FROM keyword indicates which table you are viewing (there's only one at the moment), and Agents is the name of that table. SQL keywords are traditionally typed entirely in uppercase letters.

Let's say you had to quickly set up a mission that involves explosives (like all good missions). You might want to get a list of all your agents and their specialties. You could use this query to generate that list:

```
SELECT CodeName, Specialty FROM Agents
```

As you can see from this query, you can indicate more than one field to display at a time. Figure 11.20 shows the results of this second query.

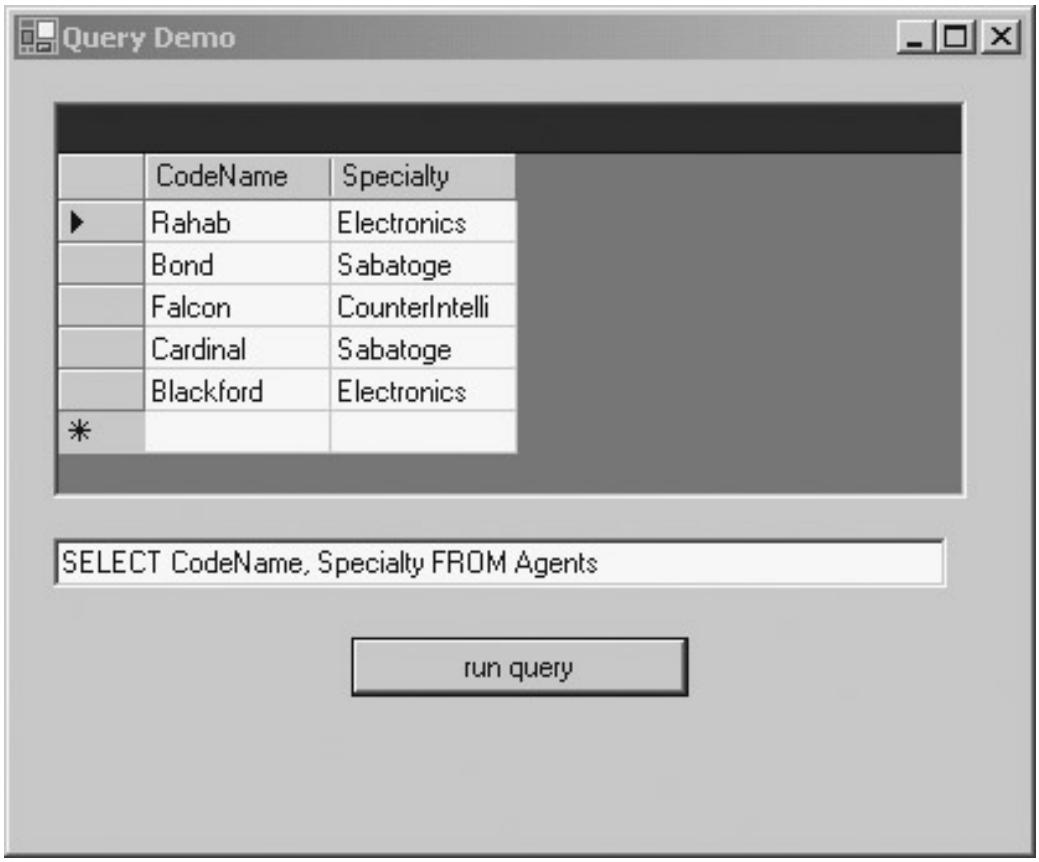


Figure 11.20: This query returns a list of the codenames and specialties.

The queries you've seen so far indicate how you can limit the number of columns shown in a query result. If you're only concerned about those spies who can handle explosives, you're actually more interested in limiting the number of rows (records) that are displayed. You can modify a SELECT statement so it only returns records that satisfy some sort of condition. Try this SQL statement:

```
SELECT * FROM Agents WHERE Specialty = 'Explosives'
```

The WHERE statement indicates you want to limit the number of rows displayed. It is followed by a condition. Conditions in SQL are similar to the ones you have used many times in C#, but there are some differences. First, the condition almost always compares a field name to a value. Second, SQL conditions use a single equals sign (=) for equality rather than the double equals (==) used for equality in C#. Finally, string values in SQL are encased in single quotes rather than the double quotes you are used to in C#. The result of this query is shown in Figure 11.21.

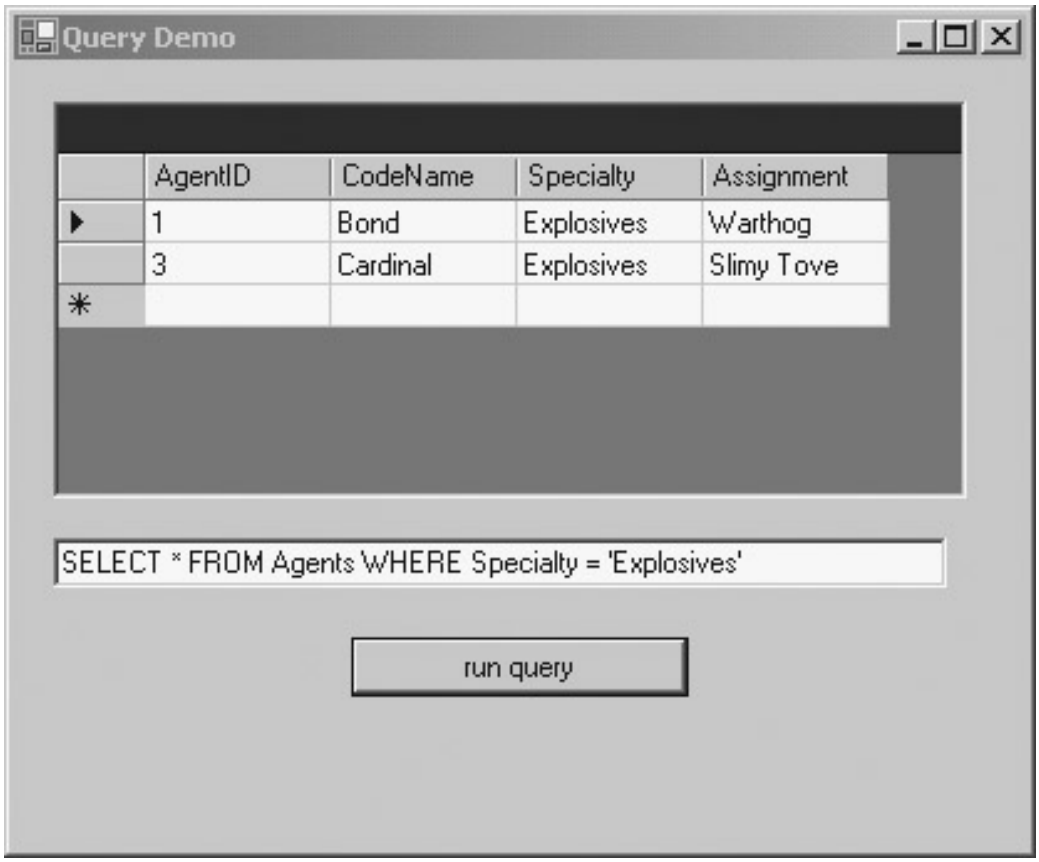


Figure 11.21: The query returns all the fields, but only the records of the spies who specialize in explosives.

Of course, you can combine both types of queries. If you want a list of the code names and assignment of spies who specialize in explosives, you can use the following query:

```
SELECT CodeName, Assignment FROM Agents WHERE Specialty =
'Explosives'
```

The results of this query are illustrated in Figure 11.22.

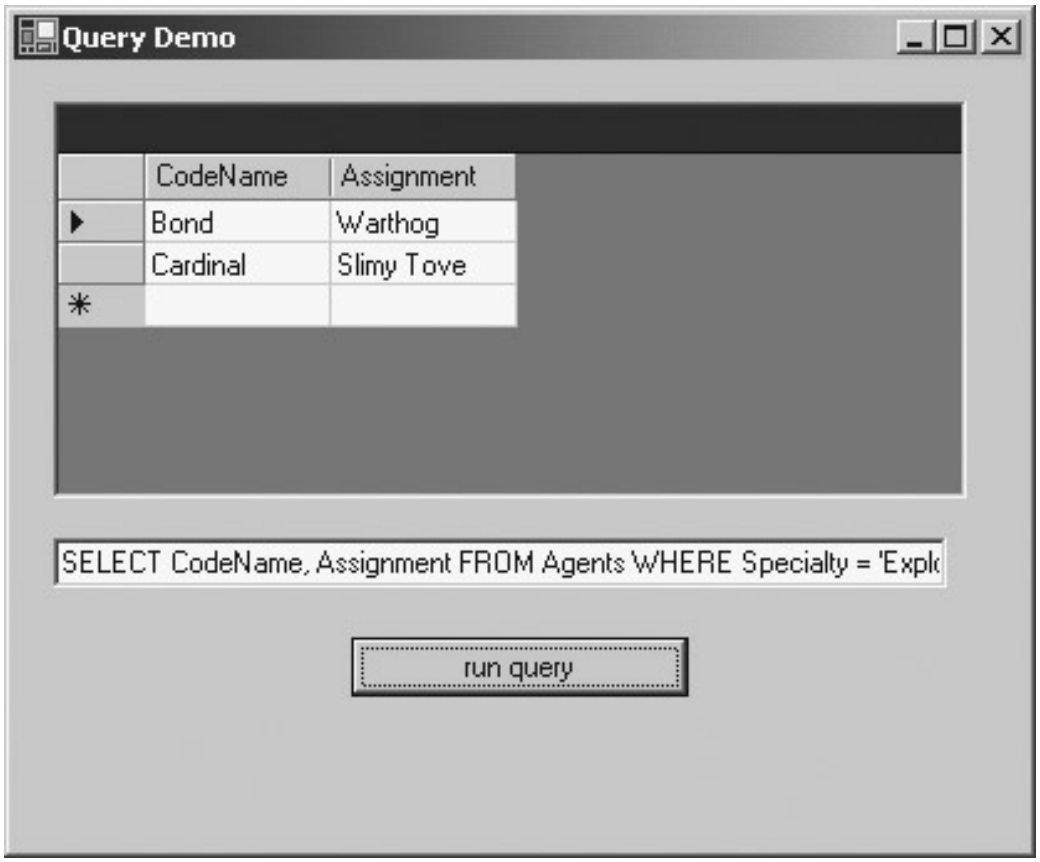


Figure 11.22: This query limits both the number rows and the number of columns.
In essence, the SELECT statement is used to filter the data to answer some kind of question.

Trap I allowed the user to type in a SELECT statement here simply to keep the code simple. As you can imagine, real users don't usually know much about SQL, so they probably would really mess this program up. It would be better to build some sort of visual mechanism that creates the SQL statement on the fly than to let the user directly enter SQL. You'll see some examples of that later in the chapter.

Using an Existing Database

The Query Demo program is based on the existing SimpleSpy database.

I already created the Simple Spy database, so I don't have to make it again. Simply start a new project, open the server explorer, and the Simple Spy database will still be there. The databases are a function of the entire machine. They are not necessarily tied to any particular project. Drag the Agents table onto the new form, and rename the resulting SqlDataConnection and SqlDataAdapter. Create a new data set from the DataAdapter, and add a data grid to the form. Your form should look something like Figure 11.23.

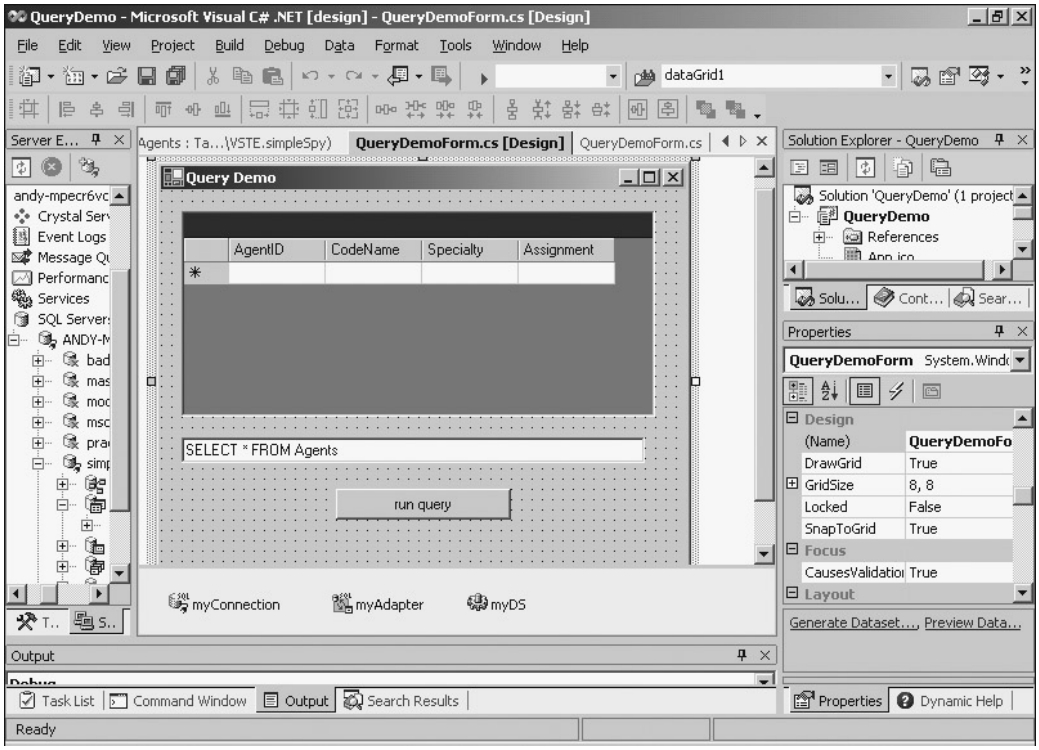


Figure 11.23: The query demo has been attached to the SimpleSpy database.

To finish the connection, set the data grid's datasource to myDS, and its data member to Agents. Finally, fill the data set in the form's load method:

```
private void QueryDemoForm_Load(object sender,
        System.EventArgs e) {
  myAdapter.Fill(myDS);
} // end form load
```

Hint In the last few steps, you have recreated the entire simple spy program. It's interesting that a procedure that took 20 minutes or more the first time can be done in just a few minutes the second time. With a little practice, you'll be able to completely hook up a database in a minute or two.

Adding the Capability to Display Queries

The visual layout of the form is not challenging. It contains a data grid called dgSpies, a textbox called txtQuery, and a button called btnQuery. All the action happens in the btnQuery click event:

```
private void btnQuery_Click(object sender,
System.EventArgs e) {
  DataSet qDS = new DataSet("results");
  myAdapter.SelectCommand.CommandText = txtQuery.Text;
  myAdapter.Fill(qDS, "results");
  dgSpies.SetDataBinding(qDS, "results");
} // end btnQuerey
```

The code begins by creating a new DataSet object called qDS (for query data set). This new data set has one table, called results. The qDS data set holds the results of the query in its results table. Remember, a data set is a local copy of a database. It must be filled from some sort of data adapter. The new data set gets a subset of the original database. The data adapter class has a SelectCommand property that holds a command for selecting the data. This property holds an SQLCommand object, which has a CommandText property. I can set the CommandText property to a SELECT statement, and this applies to all subsequent fill statements related to the adapter. I use

the adapter's Fill() method to fill qDS with the filtered data. Finally, I used the SetDataBinding() method of the data grid to apply the new Data Set to the data grid and display the results.

Letting the user enter information in a text box is just too risky, so I added an exception handling routine so the program won't choke up if the user enters a query in some odd format (like "get me all the spies who do sabotage.")

```
DataSet qDS = new DataSet("results");
  myAdapter.SelectCommand.CommandText =
txtQuery.Text;
  try {
    myAdapter.Fill(qDS, "results");
    dgSpies.SetDataBinding(qDS, "results");
  } catch (Exception exc){
    MessageBox.Show("Something went wrong");
    myAdapter.SelectCommand.CommandText =
"Select * from Agents";
    myAdapter.Fill(qDS, "results");
    dgSpies.SetDataBinding(qDS, "results");
  } // end try
} // end btnQuery
```

Creating a Visual Query Builder

It isn't reasonable to expect your users to know how to build a SELECT statement. Instead, you may want to prepare some predefined queries for the user, or you might want to build a visual query builder. Figures 11.24 and 11.25 illustrate a program that enables the user to build queries dynamically.

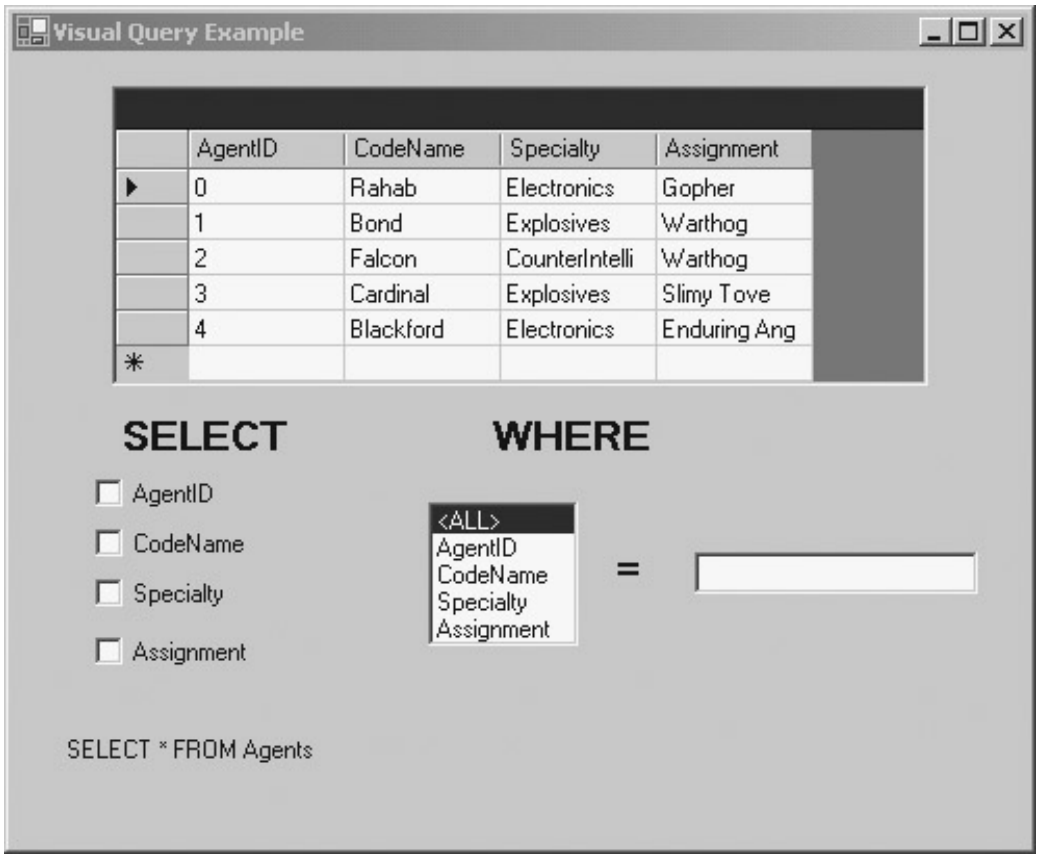


Figure 11.24: The form is just like QueryDemo, but it adds some other controls for building the

query.

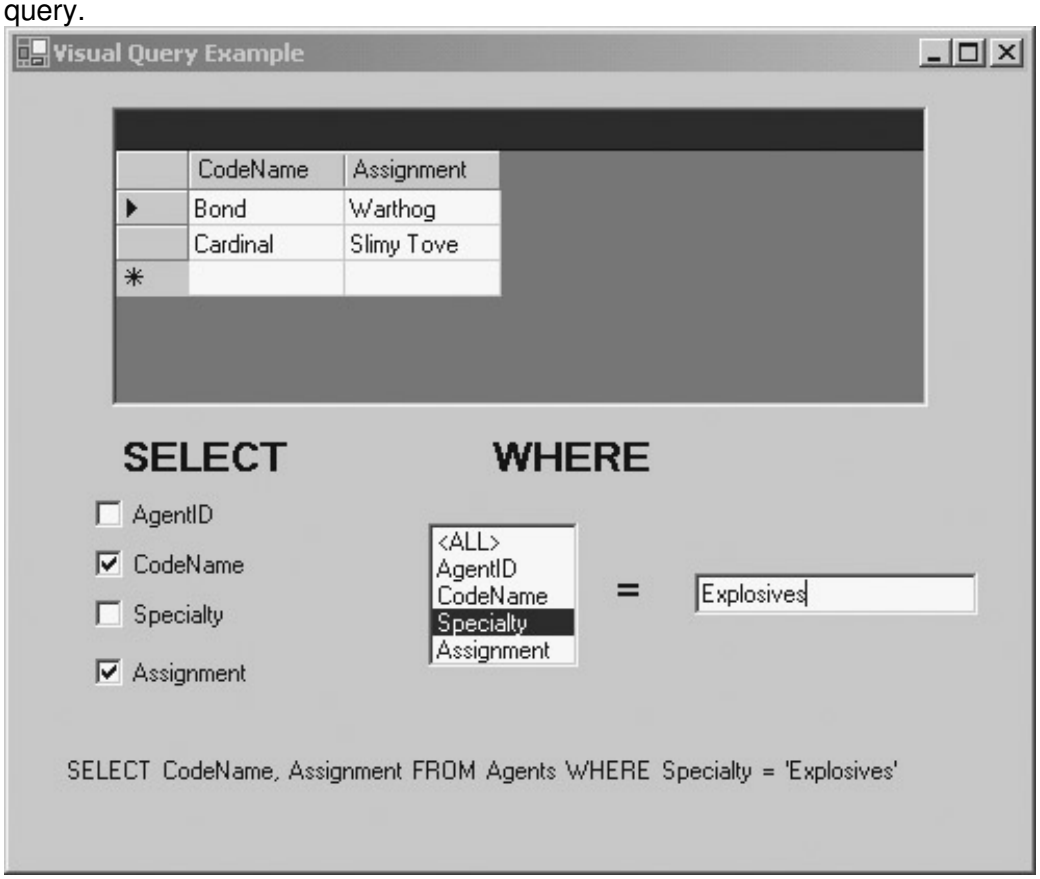


Figure 11.25: If the user types a value into the textbox and chooses a field from the listbox, only records that satisfy the resulting condition are displayed.

The program displays the SELECT statement as it is being built. The query is automatically updated every time any of the screen elements is changed. This program is very easy to use, and it provides a great deal of power to the user.

Creating the Layout for the Visual Query Builder

The Visual Query program uses a number of controls. The most prominent of these is a data grid, which is bound to a data set as you have done several times in this chapter. To create the bound data grid, follow these steps:

1. Locate your database in the server explorer pane.
2. Drag the Agents table to the form.
3. Rename the resulting DataConnection and DataAdapter Objects.
4. Create a data set from the DataAdapter (using the link at the bottom of the property menu).
5. Rename the resulting instance of the data set.
6. Set the DataSource property of the data grid to the data set.
7. Set the DataMember property of the data grid to the Agents table.
8. Call the data adapter's fill method to fill the data set in the form's load event.

Hint It might seem crazy to attach to the same database three different times, but it's really good practice. Repetition is the best way to ensure you know how to attach to a database. Every data application you write starts out something like this, so it's a good skill to rehearse.

Figure 11.26 shows the Visual Query form in the designer.

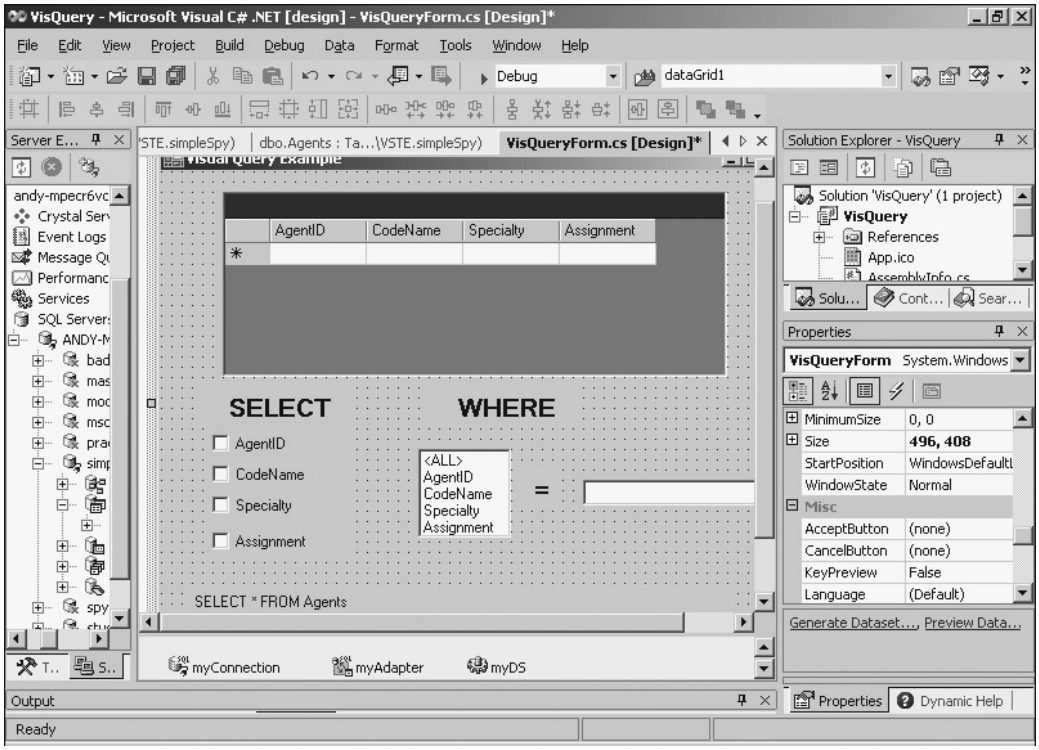


Figure 11.26: The Visual Query form features checkboxes, list boxes, and a couple of labels.
The check boxes have predictable names (chkAgentID, for example). The list box is called lstField, and I loaded the field names into the list box in design time. The text box is called txtMatch, and the label holding the query is called (cleverly enough) lblQuery. The only class–level variable is a string called theQuery.

Initializing the Code in the Load Event

As usual, there is some initialization in the form's load event:

```
private void VisQueryForm_Load(object sender,
        System.EventArgs e) {
  myAdapter.Fill(myDS, "Agents");
  lstField.SelectedIndex = 0;
} // end form load
```

The load event has only two tasks. First, it fills up the data set from the adapter. Second, it selects element 0 in the list box, so there is always some element selected in the list box.

Building a Query

All the elements on the form cooperate to build a SELECT statement (query). Every time the user changes any element (except for the data grid itself) the component fires off a call to the buildQuery() method. This method is essentially a general–purpose event handler called by most of the screen components.

> **Hint** The actual event that triggers the buildQuery() method changes based on the component type, but they were all pretty easy to guess. For example, I called buildQuery() on the CheckChanged event of the check boxes, the SelectedIndexChanged event of the list box, and the TextChanged event of the text box. When I had a "build query" button, I had its Clicked event attached to the buildQuery() method as well.

Building the query is a little more involved than it might seem, so I broke it into functions. The SELECT part of the query comes primarily by examining the status of the check boxes. The getSelect() method builds the theQuery string based on which check boxes are currently checked. The getWhere() method takes theQuery and adds a WHERE clause based on which element in the list box is selected and what value is currently in the text box. By the time getWhere() is finished running, theQuery has a legal SQL SELECT statement that can be used to update the data set.

```
private void buildQuery(object sender,
System.EventArgs e) {
  getSelect();
  getWhere();
  lblQuery.Text = theQuery;
  runQuery();
} // end buildQuery
```

After returning from the getWhere() method, the program copies the value of theQuery to the label.

Displaying the Query

It isn't really necessary to display the query, but I did so for two reasons. Most importantly, the entire program is centered around building a query, so having the query visible all the time gave me a quick indication if my algorithms were correct. (They weren't at first, but with enough feedback and tweaking, I was able to fix them). While I was testing the program, I took the call to runQuery() out of the buildQuery() method altogether, and only ran runQuery() when I pressed a button. Most of the time, I could tell just by looking at the query whether it would work or not. Once I got the algorithm right, I added the call to runQuery(), because the user will always want to see the results of the query.

There's nothing wrong with providing feedback to the user. If the user already knows some SQL, the statement will be familiar. If not, this basic form of SQL is close enough to English to make sense even to non–technical users.

Getting the SELECT Clause from the Check Boxes

The SELECT clause is used to determine which fields should be displayed. Check boxes seem like a good option for this kind of input, because any number of them can be selected.

```
private void getSelect(){
      //create the SELECT part of the
query from check boxes
      theQuery = "SELECT ";
      bool none = true;

      if (chkAgentID.Checked){
        theQuery += "AgentID, ";
        none = false;
      }  // end if
      if (chkCodeName.Checked){
        theQuery += "CodeName, ";
        none = false;
      } // end if
      if (chkSpecialty.Checked){
        theQuery += "Specialty, ";
        none = false;
      } // end if
```

```
      if (chkAssignment.Checked){
        theQuery += "Assignment, ";
        none = false;
      } // end if
      if (none){
        theQuery += "*, ";
      } // end if

      //remove the last comma
      theQuery = theQuery.Substring(0, theQuery.Length -2);

      theQuery += " FROM Agents";
    } // end getSelect
```

The method begins by setting the value of theQuery to "SELECT ". All of the queries created by this program begin with that word. I then created a Boolean variable called none and initialized it to true. The none variable is used to indicate that none of the check boxes have been checked.

The method then looks at each check box to see if it is checked. If so, none is false (because at least one of the check boxes has been checked) and the name of the current field is appended to theQuery. If, after all the check boxes have been examined, the value of none is still true, I use the asterisk (which stands for all fields) as a field name.

Notice that all the field names end with a comma. SQL requires the list of fields to be separated by commas. It's easy to put a comma after every field, but SQL does not want a comma after the last field. The problem is determining which field is last. I used a little string manipulation magic to solve this problem. I added a comma after every field name, and then once the field names were finished, I always removed the last two characters from the string, eliminating the last comma and space. (Sneaky, huh? You've got to be sly when you're working with secret agents all the time.) In this program, all queries come from the Agents table, so I just added "FROM Agents" to the end of the query.

To make sure you understand how this is working, run the Visual Query program for a while and take a careful look at the query as you select and deselect check boxes.

Getting the WHERE Clause from the List Box and Text Box

The WHERE clause of an SQL statement is usually used to compare the name of a field with some possible value. To automate the creation of this part of the SQL statement, I used a list box to select field names and a text box to input matching values.

```
private void getWhere(){
  //Create WHERE clause of query from list box
  if (lstField.SelectedIndex != 0){
    theQuery += " WHERE ";
    theQuery += lstField.Text;
    if (lstField.Text != "AgentID"){
      theQuery += " = '";
      theQuery += txtMatch.Text;
      theQuery += "'";
    } else {
      theQuery += " = ";
      try {
        int temp = Convert.ToInt32(txtMatch.Text);
        theQuery += Convert.ToString(temp);
      } catch (Exception exc) {
        theQuery += "0";
      } // end try
```

```
    } // end agentID if
  } // end something selected if
} // end getWhere
```

In most situations it is very easy to build the WHERE clause, because it's simply a matter of extracting the list box text (which indicates the currently selected field) and the text from the text box (which indicates the value the user is trying to find). However, there are two special cases. If the selectedIndex property of the list box is zero (indicating no WHERE clause) the rest of the method is skipped, because the user doesn't want to include a condition. If the user selected "AgentID" things are slightly more complicated because the text box returns a text value and AgentID requires an integer. The query is still a string, but you do not need single quotes around integer values. For example, "SELECT * from Agents WHERE AgentID = '0'" will not work, but "SELECT * from Agents WHERE AgentID = 0" will. There are two problems to solve. First, the text box value will not be surrounded by single quotes (this one is easy to fix). The second problem is what to do if the user chooses the AgentID field while a non−numeric value is in the text box. I used some exception handling sleight−of−hand to fix this. I try to convert the text from the text box into an integer and store it in a temporary variable. I'll then convert that integer back into a string and add it to the query. If there was an exception (which happens if the original value of the text box cannot easily be converted into an integer), I simply add the value 0 to the query, which is guaranteed to be legal.

Running the Query

The buildQuery() method (and its offspring getSelect() and getWhere()) do a good job of generating a legal SQL query in theQuery. The runQuery() method simply updates the data set and data grid based on the new query.

```
private void runQuery(){
  //runs the current query
  DataSet qDS = new DataSet("results");
  myAdapter.SelectCommand.CommandText = lblQuery.Text;
  myAdapter.Fill(qDS, "results");
  dgSpies.SetDataBinding(qDS, "results");
} // end runQuery
```

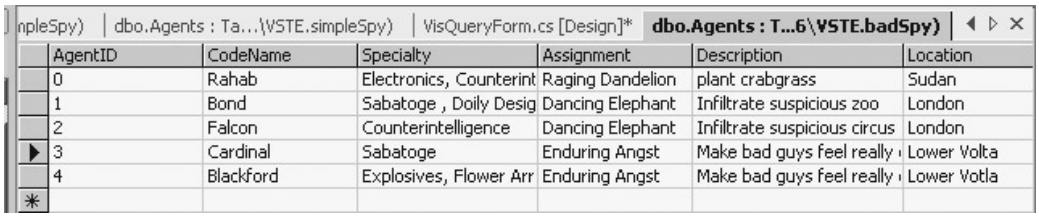
The first task is to create a new data set called qDS with one table called results. I then set up the Select command to the text of lblQuery. I then use the Fill() method of the adapter to fill the data set from the adapter, and bind the data grid to the new data set.

Working with Relational Databases

The simple spy database does the job, but it is incomplete. Any self−respecting spymaster would keep the following information on each agent:

- CodeName
- Specialty (each spy could have several specialties)
- Assignment
- Assignment description
- Location

It might be tempting to build a slightly bigger table to hold this information. Figure 11.27 illustrates such a table.



AgentID	CodeName	Specialty	Assignment	Description	Location
0	Rahab	Electronics, Counterint	Raging Dandelion	plant crabgrass	Sudan
1	Bond	Sabatoge , Doily Desig	Dancing Elephant	Infiltrate suspicious zoo	London
2	Falcon	Counterintelligence	Dancing Elephant	Infiltrate suspicious circus	London
3	Cardinal	Sabatoge	Enduring Angst	Make bad guys feel really	Lower Volta
4	Blackford	Explosives, Flower Arr	Enduring Angst	Make bad guys feel really	Lower Votla

Figure 11.27: This version of the spy database has more information, but it also introduces a number of problems.

When you carefully analyze this version of the spy database, you'll notice a couple of problems that crop up frequently in real databases. First, many of the spies have multiple talents (my personal favorite is explosives and flower arranging). It will be difficult to write a query that finds an agent with a flower arranging skill, because the only agent with that skill also has explosives listed in the same field. (You know, flower arranging can be a deadly art in the hands of a master practitioner…) There are other problems. The description and location fields tend to be closely related to the assignment field. That makes sense, because it is supposed to be a description of the assignment, and each assignment has only one location. However, there are some inconsistencies. Does Operation Dancing Elephant take place in a circus or a zoo? Because the description of the assignment was typed in two different places in the database, there is conflicting information about the operation. Likewise, Operation Enduring Angst might be in Lower Volta, or it might be in Lower Votla. Although these examples are deliberately outlandish, the problems they point out are real. Many databases have variations of these same weaknesses. The answer to better–behaved databases is a practice called *data normalization*.

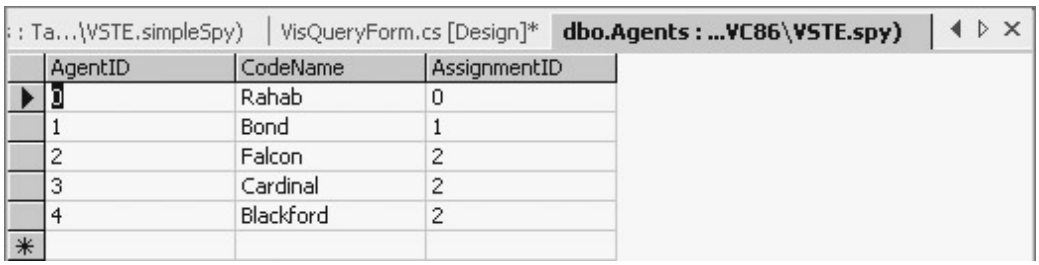
Improving Your Data with Normalization

Data normalization can (and does) take up entire books, but it can be summarized by a list of simple rules:

- Break your data into multiple tables
- No field can have a list of entries
- Do not duplicate data
- Each table describes only one entity
- Each table has a single primary key field

As an example of data normalization in action, I built one more version of the spy database. It retains the ability to return all the data needed, but it avoids some of the pitfalls of the single–table database.

First, take a look at the improved version of the Agents table, featured in Figure 11.28.

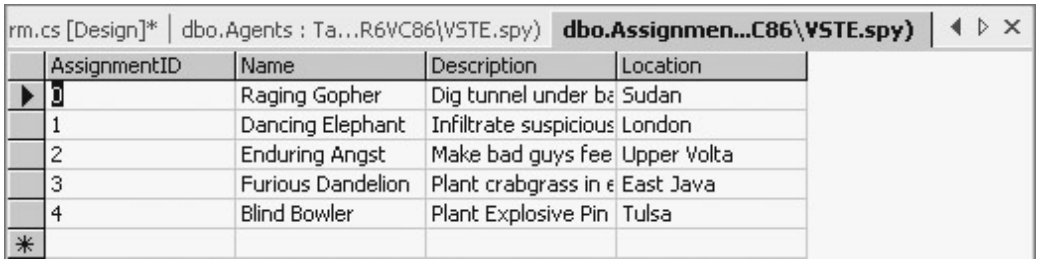


AgentID	CodeName	AssignmentID
0	Rahab	0
1	Bond	1
2	Falcon	2
3	Cardinal	2
4	Blackford	2

Figure 11.28: The Agents table is quite a bit simpler than it was before.

You might be surprised how little information remains in the Agents table. The Assignment, Specialty, Description, and Location fields have totally disappeared from the table. (Don't worry, they'll reappear shortly.) The only remaining fields are AgentID, CodeName, and AssignmentID. The Assignment ID field contains only numeric values. The number in the AssignmentID field is

used to look up a record in another table, illustrated in Figure 11.29.



AssignmentID	Name	Description	Location
0	Raging Gopher	Dig tunnel under ba	Sudan
1	Dancing Elephant	Infiltrate suspicious	London
2	Enduring Angst	Make bad guys fee	Upper Volta
3	Furious Dandelion	Plant crabgrass in e	East Java
4	Blind Bowler	Plant Explosive Pin	Tulsa

Figure 11.29: The Assignments table describes all the information related to a specific operation.

I built the Assignments table by taking a careful look at the data in my original expanded database. In a properly normalized database, all of the information in a table describes one type of entity. On closer examination Figure 11.27 (the bad spy table) has fields that describe two different kinds of information. The AgentID, CodeName, and Specialty fields describe an *Agent*, but the Assignment, Description, and Location fields refer to the *Operation* the agent is assigned to. (Specialties are an entirely different problem, and I'll describe them later on.) The Agents table still needs a way to determine which operation an agent is on, but it is redundant to describe each operation's details for each agent on the assignment. For the purpose of this example, each operation has one name, one assignment, and one location. The information that pertains to the Assignment is placed in the separate Assignments table. The Assignments table also has a primary key, so each assignment has a unique key. The Agents table has an AssignmentID field, which contains only a reference to the key field of the Assignments table.

Hint A field that contains the primary key of another table is called a *foreign key reference* in most database applications.

Note that the AssignmentID field appears in both the Agents table and the Assignments table. It isn't necessary to give these two fields the same name, but it makes the next step easier.

Using a Join to Connect Two Tables

SQL Server (even the simplified version included with Visual Studio) includes a visual tool to help connect two tables. Each Database in the SQL Server list in the server explorer has a *Data Diagrams* option. Right–click on the Data Diagrams element and choose New Data Diagram in the same way you created new tables. You see a screen that lets you add tables. Choose both the Agents and Assignments tables. You see a graphic representation of the tables that looks like Figure 11.30.

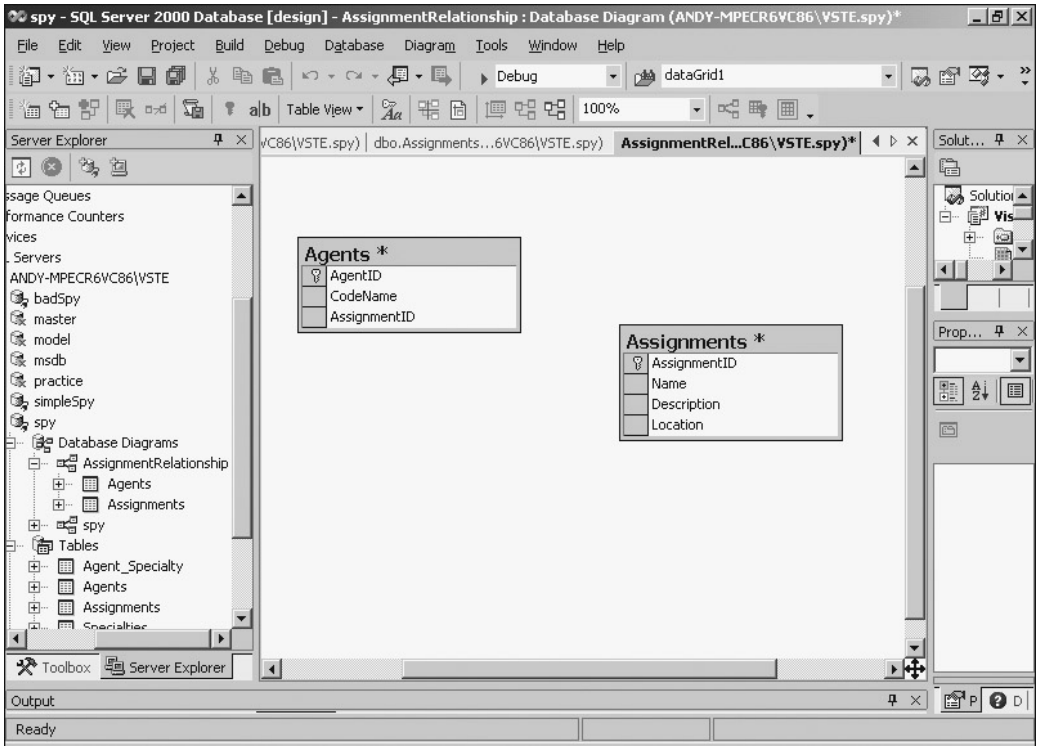


Figure 11.30: The data diagram tool shows the tables in your database.

It's important to indicate there is a relationship between the AssignmentID fields in the two tables. This is easily done in the data diagram window by simply dragging the mouse from the box to the left of AssignmentID in Agents, and the similar box of AssignmentID in the Assignments table. The dialog shown in Figure 11.31 appears.

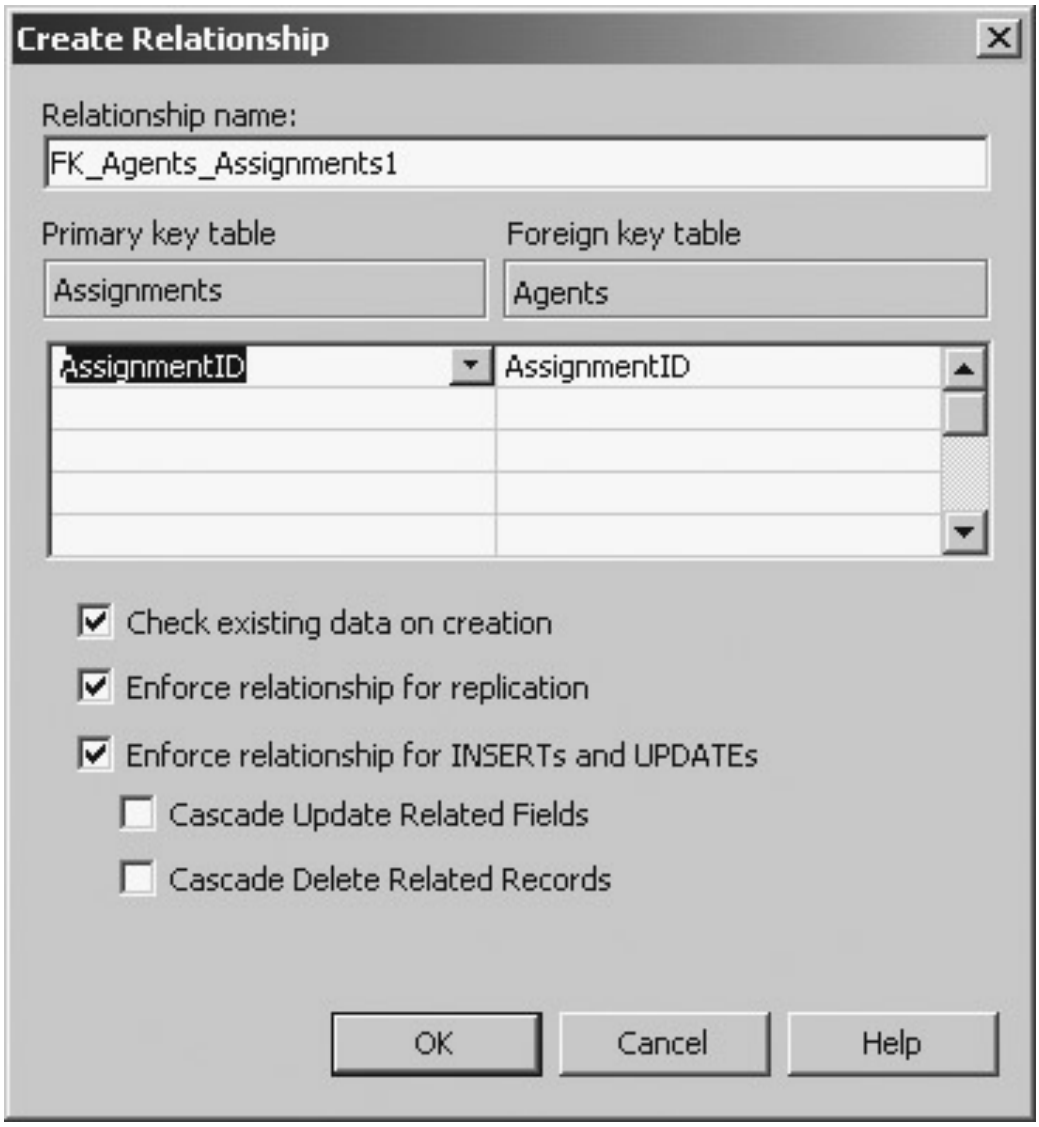


Figure 11.31: You can leave the default values of the Create Relationship dialog.

The Create Relationship Dialog helps you to define the relationship between the two tables. Relationships always involve the primary key of one table and a foreign key in another table. The Create Relationship dialog usually guesses correctly which table has the indicated key as its primary key.

Trap The two fields must have the same type, or the relationship will not be established.

After the relationship has been created, the data diagram looks something like Figure 11.32.

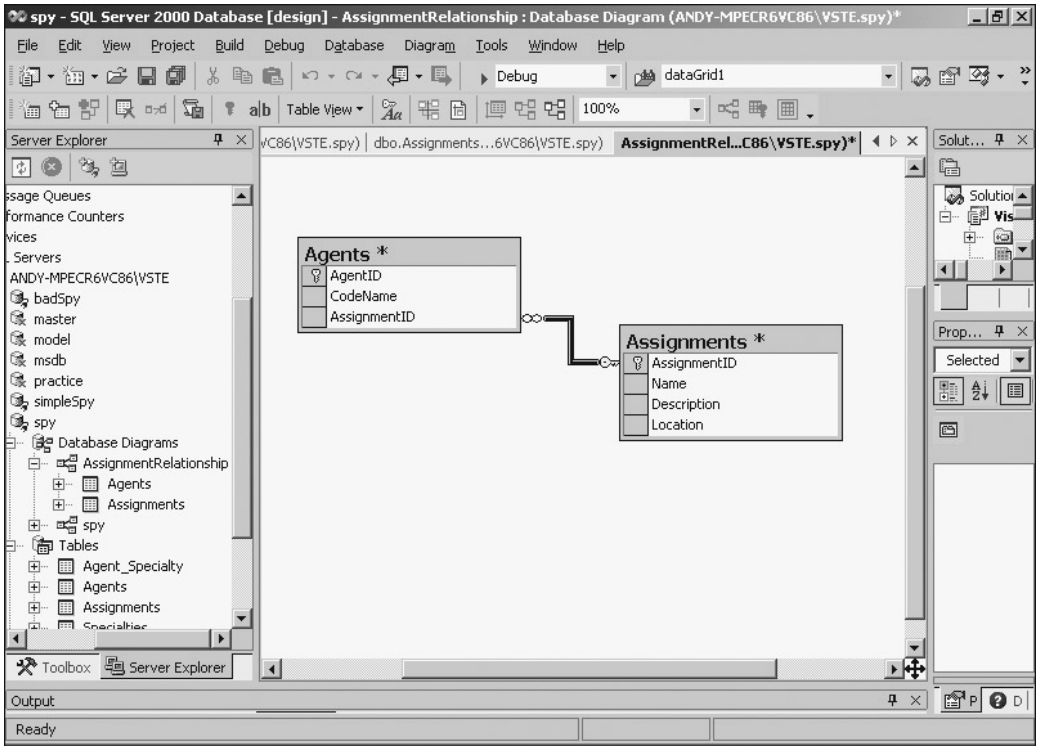


Figure 11.32: The solid line indicates the relationship between the Agents and Assignments tables.

Creating a View

The data has been normalized, which provides some important advantages, but the user really doesn't care. The user doesn't want to have to look up which assignment is assignment number 2, for example. Data management systems have a special entity called the *View* which enables you to generate a "virtual table" that recognizes the relationship you've just created. You can create a view in the server explorer just like you created a table and a data view. When you right–click on the Views item and choose Create New View, you get the opportunity to include any data tables. If you choose both tables you see a screen similar to Figure 11.33.

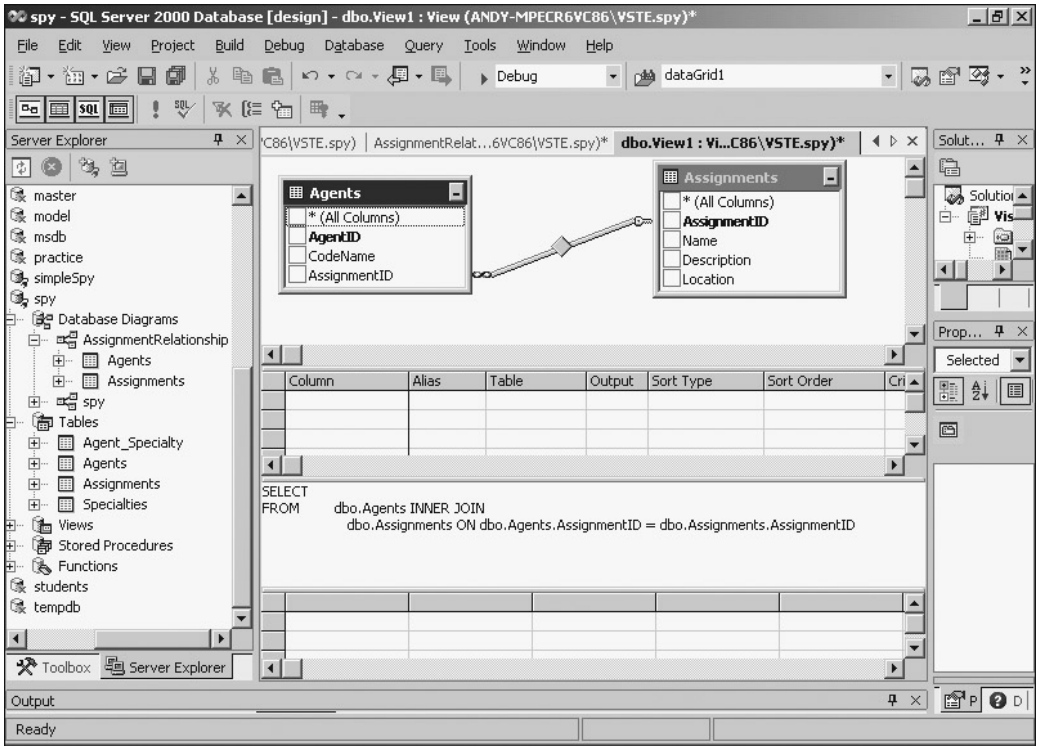


Figure 11.33: The view editor features a form of the data diagram. The relationship has been preserved.

The view editor has four main sections. The top section is a version of the data diagram. The next section is a Query By Example (QBE) tool. The third band displays an SQL statement, and the bottom layer shows the results of the current query.

Hint You might wonder why the *view* editor has all these query tools. It's because a view and a query are essentially the same thing. It's called a query when it's created on the fly in an application or by a user. A view is usually stored with the database. Often a data developer creates a number of views to reconnect any tables that have been separated by the normalization process, and any other queries that are likely to be extremely common.

The best way to understand the view editor is to look at an example. Figure 11.34 illustrates my completed Agent_Assignment View.

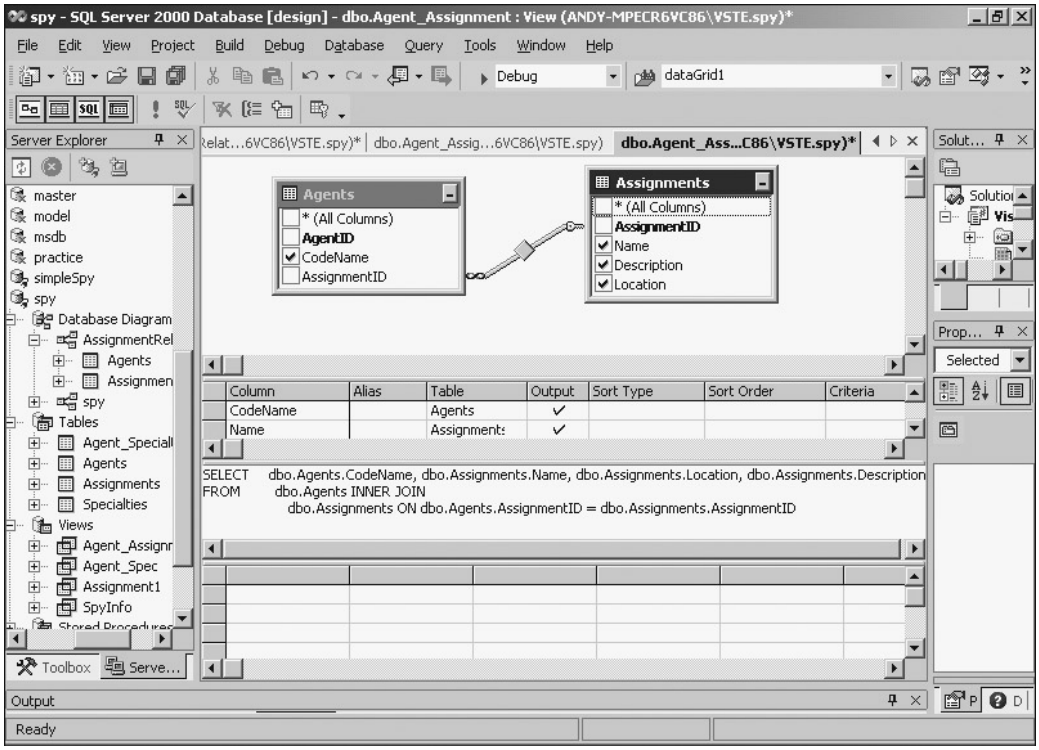


Figure 11.34: As you can see from the bottom of the screen, this view recombines the Agents and Assignments tables.

A view or query can be used to recombine tables. I created this view by using a tool called *Query By Example* (QBE). I drug all the fields I was interested in displaying from the data diagram at the top of the screen to the grid in the middle of the screen. When I did so, the center grid automatically filled with the values you see. The grid describes which fields I want to use in the query. For each field, I can assign an alias. The alias describes the name of the field in the resulting virtual table. If you do not provide an alias, the original field name is retained. You can choose whether each field is displayed. The Sort Type and Sort Order values are used to determine how the resulting data is sorted. The Criterion value enables you to set a WHERE clause for the view.

To see the results of the view, click on the exclamation point icon at the upper–left of the view editor, or choose Run from the Query menu.

> **Hint** The view editor is a great way to learn SQL syntax. It's pretty easy to experiment by dragging fields to the grid and examining the resulting SQL statement. Don't forget to periodically run the query so you can be sure that the data view you see in the bottom grid is related to the currently displayed SQL statement.

Take a careful look at the SQL statement in Figure 11.34. It can be reformatted as follows for clarity:

```
SELECT Agents.CodeName,
               Assignment.Name,
               Assignment.Description,
               Assignment.Location
FROM Agents INNER JOIN Assignments
ON Agents.AssignmentID = Assignments.AssignmentID
```

The only part of the SELECT statement that is really new is the FROM clause. The relationship between the Agents and Assignments tables is automatically represented by an INNER JOIN statement. The INNER JOIN tells the computer to display fields from the Assignment table only when the AssignmentID fields in the two tables match.

You can create SQL queries with inner joins inside your code as well as in views. The advantage of a view is the way it appears as a virtual table to the program

Referring to a View in a Program

Once you have created a view in the database, it's very simple to add that view to your programs. Simply drag the view to your form to build a data connection and data adapter for the view, then create a data set from the adapter and connect your data grid to the data set as usual.

Figure 11.35 shows a database that features a view.

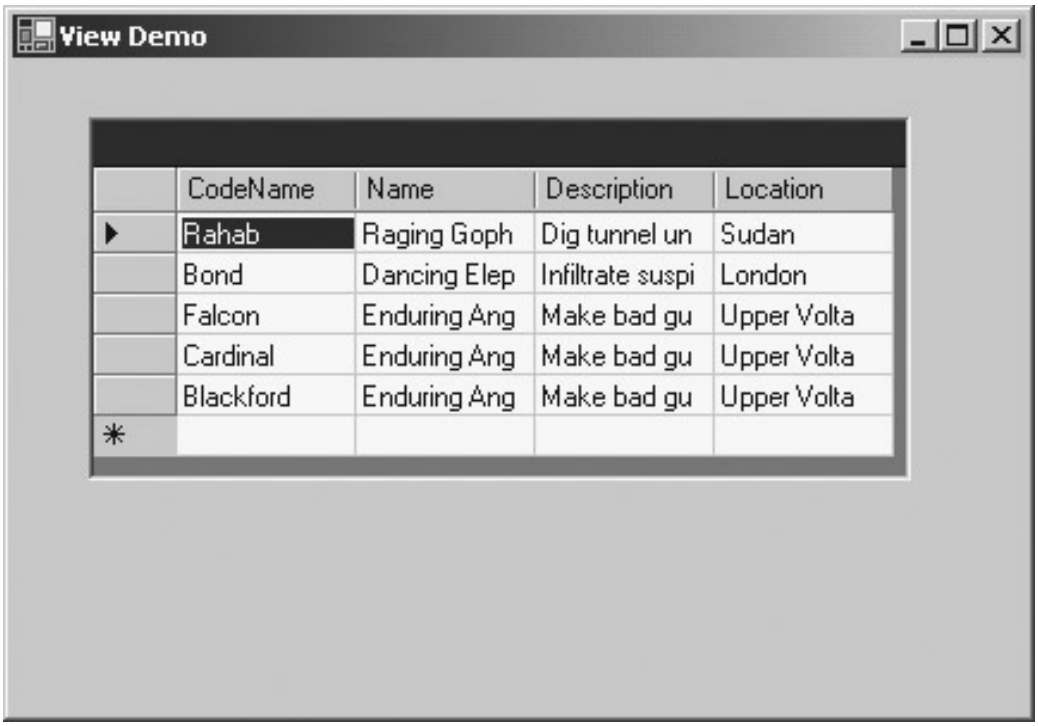


Figure 11.35: Once you've created a view, you can attach a grid to the view as if it were a table. Views give you the best of both worlds. You can design your data to improve its integrity, but the user will see the data without any cryptic foreign key values.

Incorporating the Agent Specialty Attribute

The Agent Specialty field creates another problem for the data developer. The rules of normalization indicate you should never have a list of data in a field. Each agent could have a number of specialties. If you enable the user to enter all the specialties into a single text box, you have the same ambiguity problems you prevented with the Assignments table. Also, it is hard to predict how much room to allocate for skills if the skills will be a list.

Hint The relationship between agents and specialties is called a many–to–many relationship, because each spy could have many specialties, and each specialty could belong to many different agents.

Searching for a spy with particular skills is a challenge if the skills are simply a list of text values in a field. The solution is to use *two* tables to manage the relationship between the Agent and his or her specialties. First, take a look at the Specialties table featured in Figure 11.36.

VisQueryForm.cs [Design]* | dbo.Specialtie...VC86\VSTE.spy)

SpecialtyID	Specialtyname
0	Electronics
1	Counterintelligence
2	Sabatoge
3	Doily Design
4	Explosives
5	Flower Arranging
6	Plumbing

Figure 11.36: The Specialties table simply lists all the various specialties.

The Specialties table consists of a primary key and specialty name fields. However, because each agent can conceivably have many specialties, you cannot use the same type of join that connected agents with assignments. Instead, I added another table, displayed in Figure 11.37.

sQueryForm.cs [Design]* | dbo.Specialties ...R6VC86\VSTE.spy) | dbo.Agent_Spe...C86\VSTE.spy)

Agent_SpecialtyID	AgentID	SpecialtyID
0	0	0
1	0	1
2	1	2
3	1	3
4	2	1
5	2	6
6	3	2
7	4	4
8	4	5

Figure 11.37: The Agent_Specialty table serves as a bridge between the Agents table and the Specialties Table.

The Agent_Specialty table is very interesting because the user will never see it. Instead, this table has joins to both the Agents table and the Specialties table. The Agent_Specialty.AgentID field is a foreign key reference to the Agents table, and the Agent_Specialty.SpecialtyID field is a foreign key reference to the Specialties table. The complete data diagram of the spy database including all four of its tables and their relationships is shown in Figure 11.38.

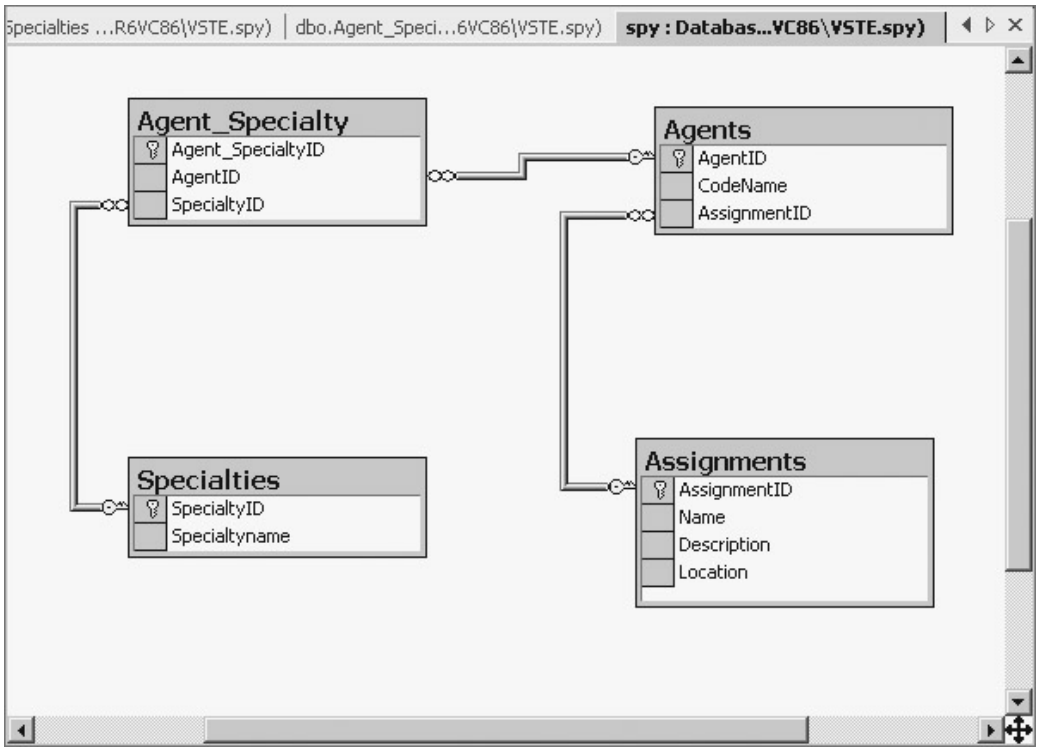


Figure 11.38: The Agents table connects directly with the Assignments table, but uses the Agents_Specialty table as a bridge to the Specialty table.
An intermediate table, such as Agents_Specialty, is frequently used to implement many−to−many relationships.

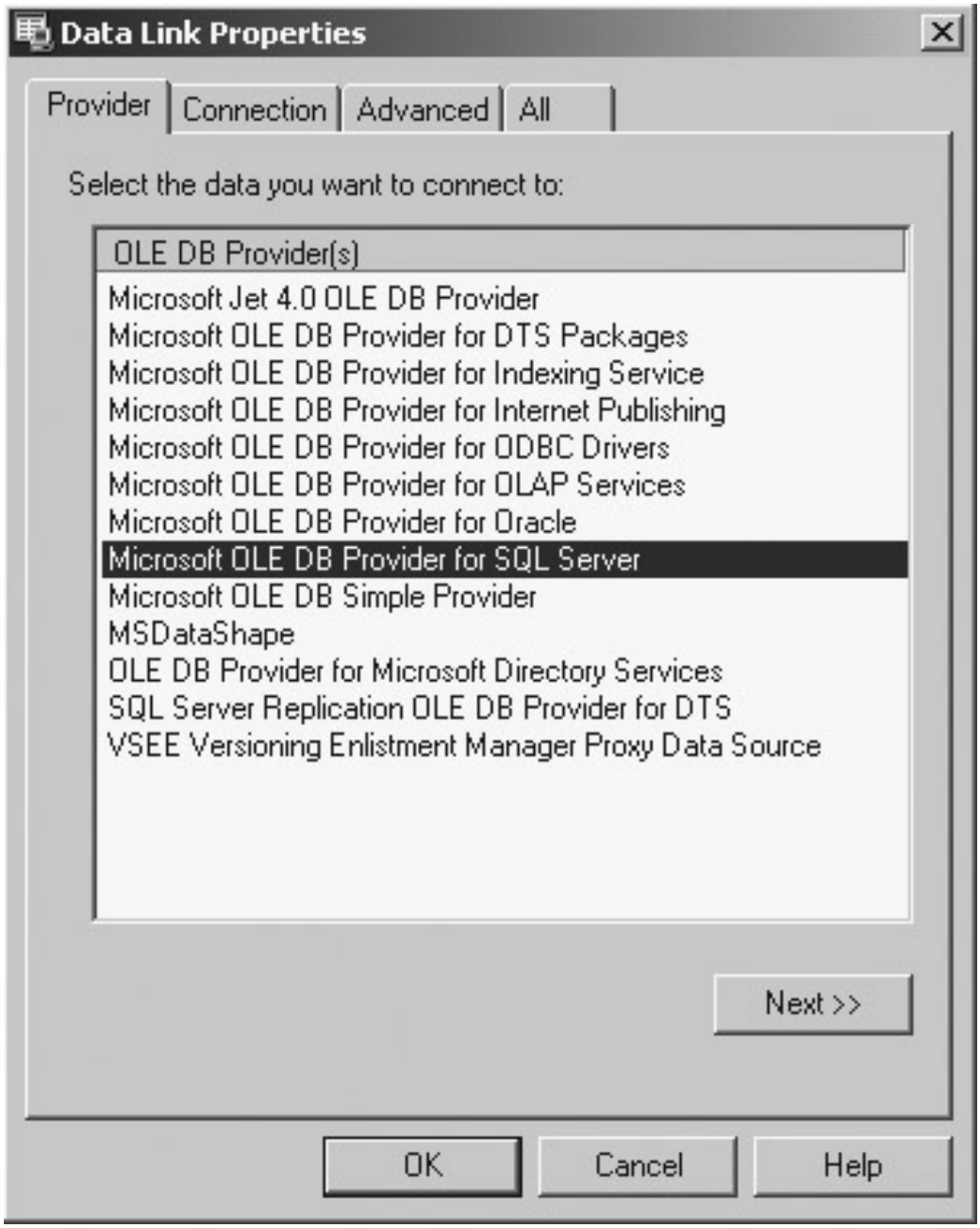
Working with Other Databases

ADO.NET is designed to work with SQL Server, and it also provides the very handy ability to connect to a number of other database formats, including XML. This is important if you already have a database that you want to build a C# program around, or if you want to use XML to exchange data with another program. The DataConnection and DataAdapter classes you have seen throughout this chapter have been designed to work explicitly with Microsoft SQL Server. However, the .NET interface also has another set of classes which can be used to attach to many other kinds of database. To illustrate, I will create a connection to an Access database and save the data as an XML file.

I began by building a simple database in Access. I decided to create a standard address book like the one you saw in Chapter 10, “Basic XML: The Quiz Maker,” showing names, addresses, and phone numbers. The database has one table called contacts.

Creating a New Connection

You still use the server explorer to connect to an existing database. Right−click on the Data Connections item at the top of the server explorer window, and choose the Add Connection menu. This produces a dialog box like the one featured in Figure 11.39.


Data Link Properties
Provider
Connection
Advanced
All
Select the data you want to connect to:
OLE DB Provider(s)
Microsoft Jet 4.0 OLE DB Provider
Microsoft OLE DB Provider for DTS Packages
Microsoft OLE DB Provider for Indexing Service
Microsoft OLE DB Provider for Internet Publishing
Microsoft OLE DB Provider for ODBC Drivers
Microsoft OLE DB Provider for OLAP Services
Microsoft OLE DB Provider for Oracle
Microsoft OLE DB Provider for SQL Server
Microsoft OLE DB Simple Provider
MSDataShape
OLE DB Provider for Microsoft Directory Services
SQL Server Replication OLE DB Provider for DTS
VSEE Versioning Enlistment Manager Proxy Data Source
Next >>
OK
Cancel
Help
Figure 11.39: The Data Link Properties Dialog lets you choose from a number of different data sources.

Trap Unlike most dialog boxes, the Data Link Properties Dialog generally pops up with the *second* (connection) tab already selected. Be sure to select the provider tab first, because the connection changes based on which type of provider you use.

Under the providers tag, choose Microsoft Jet 4.0 OLE DB Provider to connect to an Access database, then press the Next button.

Hint You also can choose a provider specific to Oracle or a number of other common data sources. If you don't see the database system you want, ODBC is a good starting point.

If you are connecting to an Access database, the dialog changes to look like Figure 11.40.

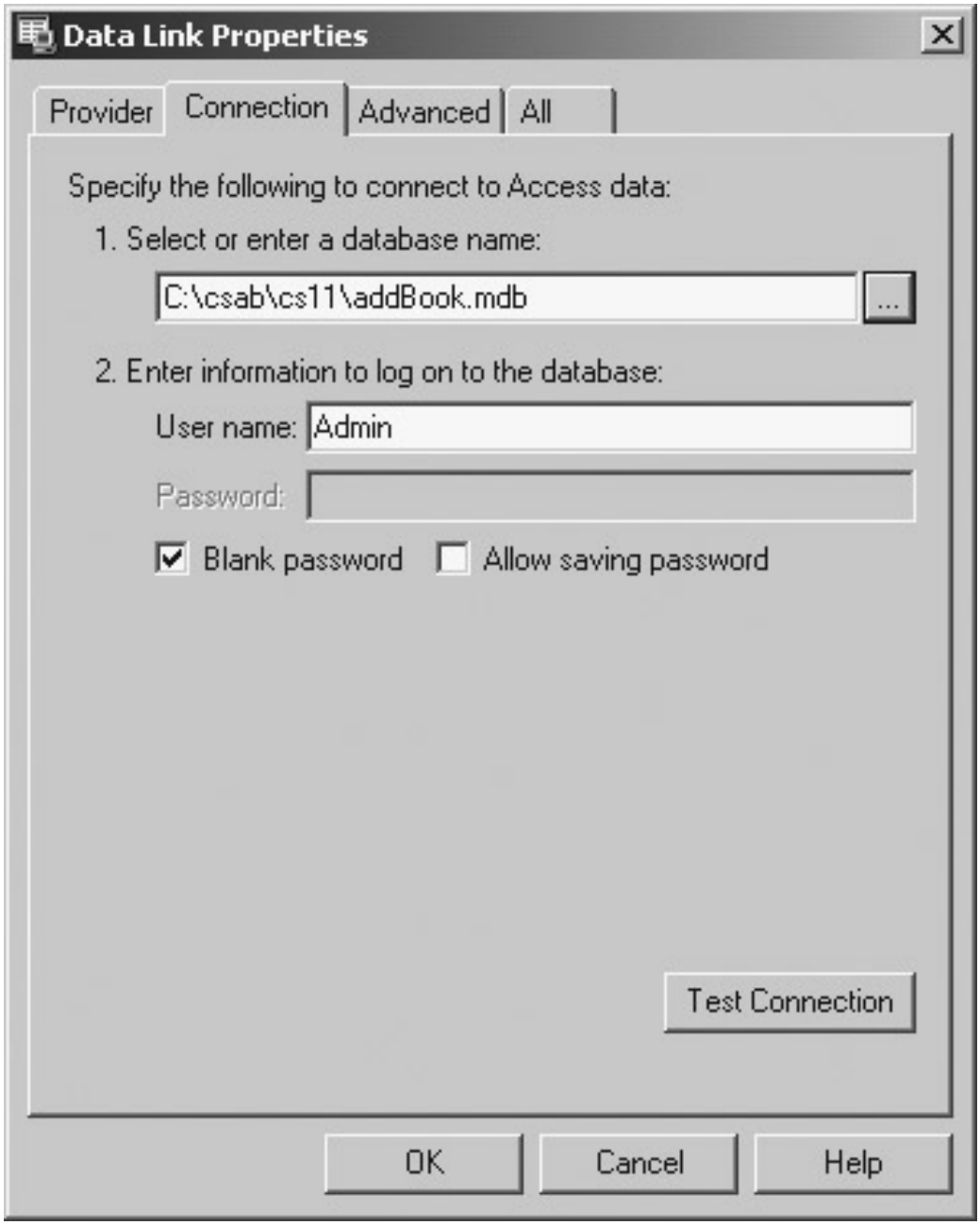


Figure 11.40: You can now select an access database from your file system.
Be sure to test the connection with the provided button. It's much easier to test the connection here than in your application, where many other things could go wrong.

For an access connection, you can simply press the OK button after choosing the data file. For other types of connections you may need to provide more, such as server account information and security information.

When you look at the server explorer, you now see the connection to the Access database. You can create a data adapter object by choosing the OleDataAdapter from the Data tab of the toolbar, as illustrated in Figure 11.41.

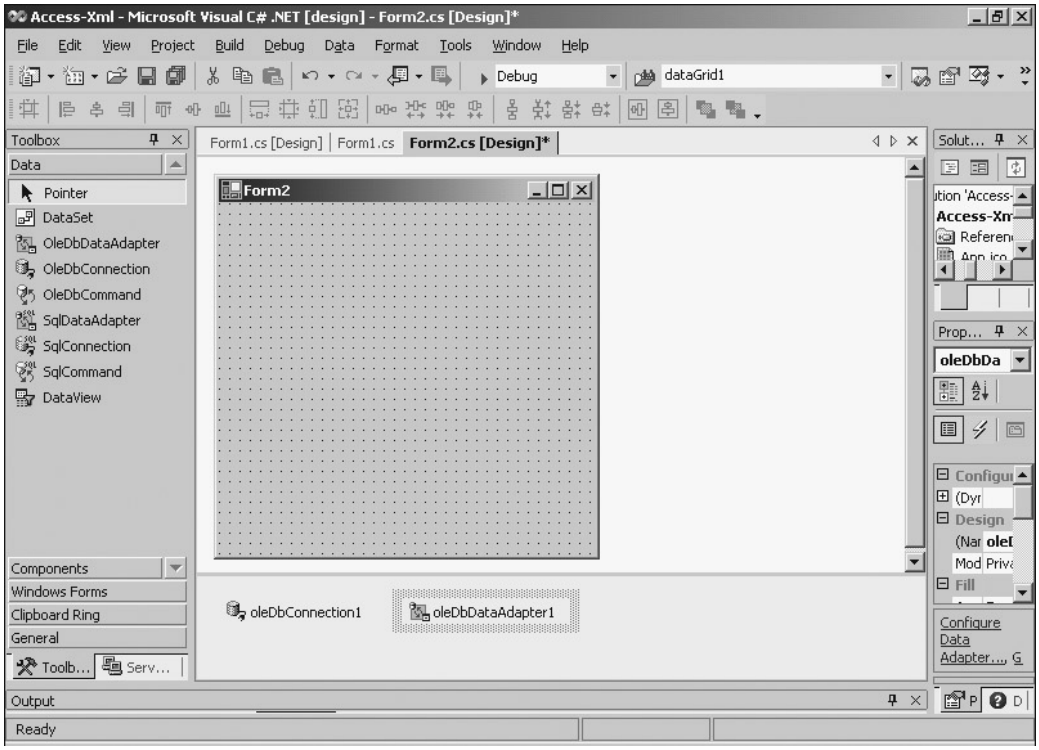
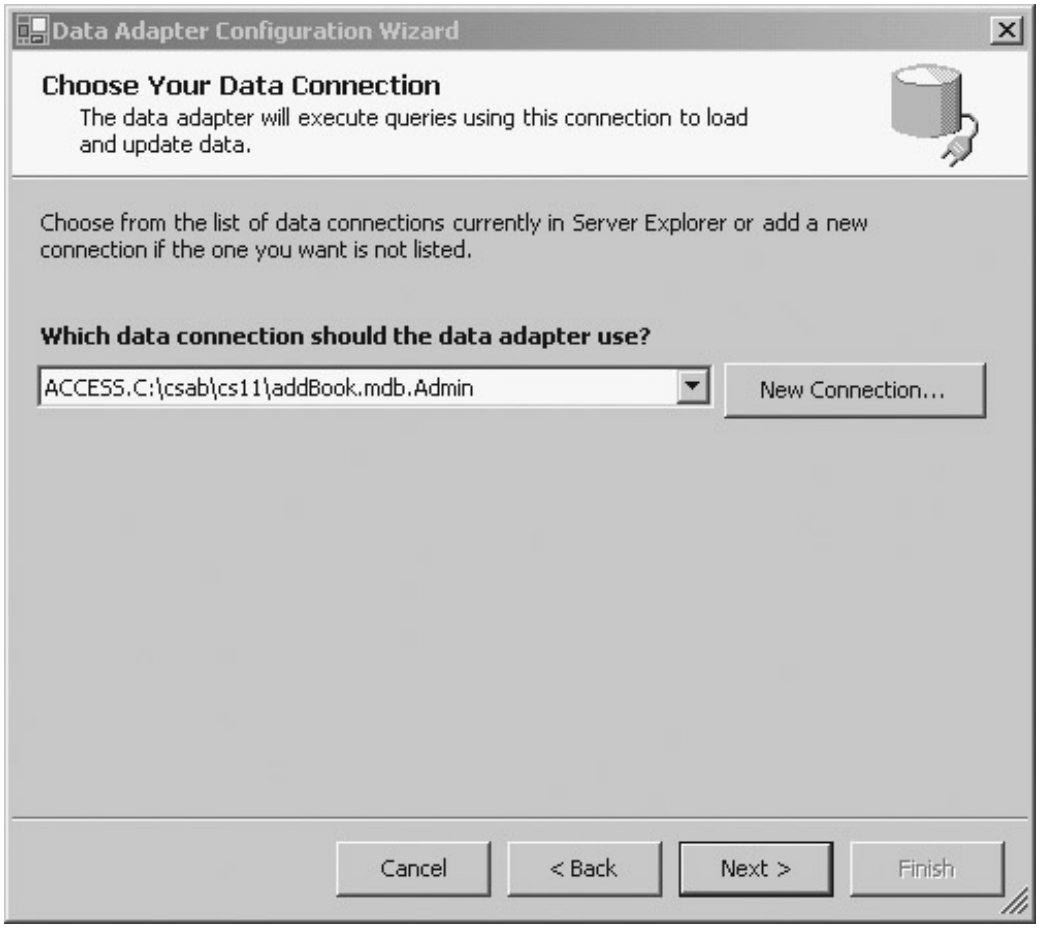


Figure 11.41: The toolbox (where you normally choose components such as textboxes and labels) has a data tab that provides access to several data components.

Drag an OleDataAdapter to the form, and the dialog shown in figures 11.42 and 11.43 appear.


Data Adapter Configuration Wizard
Choose Your Data Connection
The data adapter will execute queries using this connection to load and update data.
Choose from the list of data connections currently in Server Explorer or add a new connection if the one you want is not listed.
Which data connection should the data adapter use?
ACCESS.C:\csab\cs11\addBook.mdb.Admin
New Connection...
Cancel
< Back
Next >
Finish
Figure 11.42: All the connections established on the current machine are available from the

drop–down list.

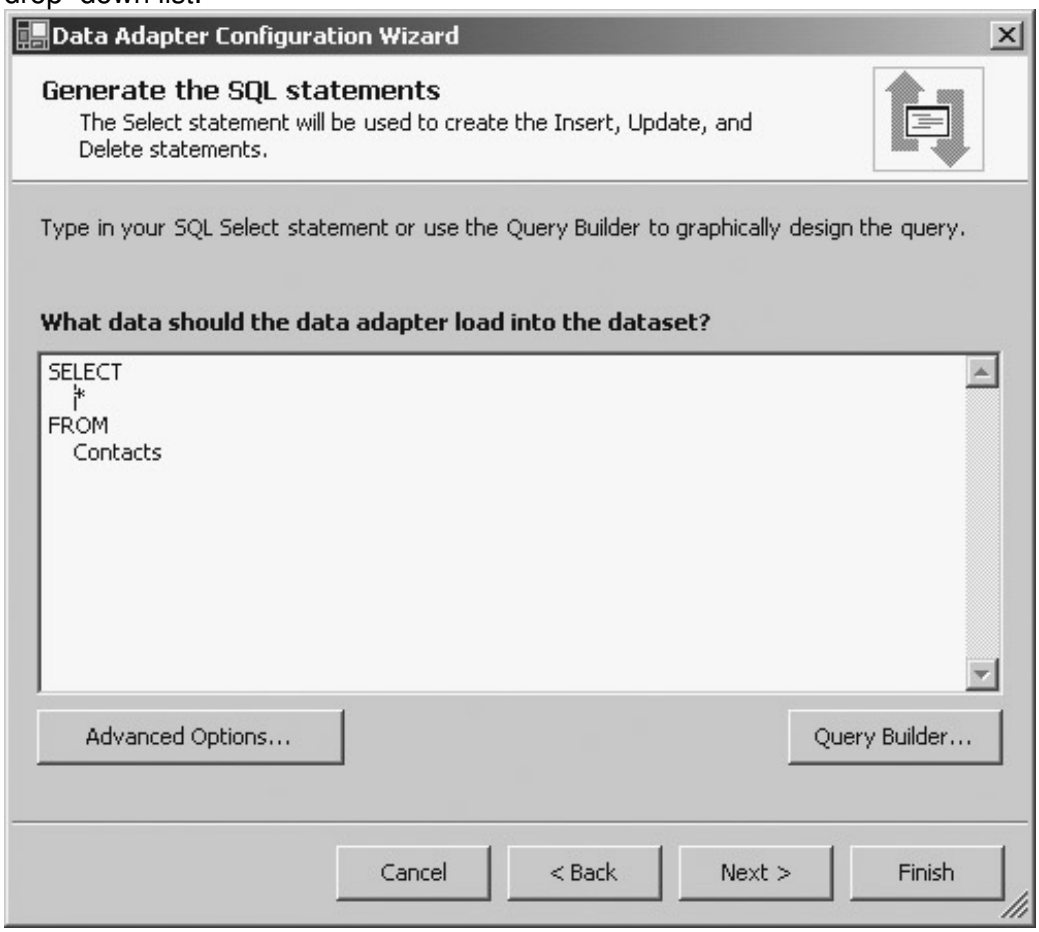

Data Adapter Configuration Wizard
Generate the SQL statements
The Select statement will be used to create the Insert, Update, and Delete statements.
Type in your SQL Select statement or use the Query Builder to graphically design the query.
What data should the data adapter load into the dataset?
SELECT
*
FROM
Contacts
Advanced Options...
Query Builder...
Cancel
< Back
Next >
Finish

Figure 11.43: You are prompted for a SELECT statement to initialize the data adapter.

The editor adds an adapter and a connection object to your form. Although these new objects are technically different objects than the SqlConnnection and SqlDataAdapter you've used, the OLE versions have the same properties, methods, and events. Encapsulation again saves you from worrying about how the internals of an SQL data object are different from an OLE data object.

Once you have an adapter and a connection, you can build a data set from the adapter, and attach the data set to a data grid just like you did with the SQL Server databases.

Converting a Data Set to XML

If you've been paying close attention, you might have noticed that every time you create a new data set Visual Studio adds a new .XSD file to your project. You might recall from Chapter 10, "Basic XML: The Quiz Maker," that an XSD file is an XML schema. The .NET framework maintains a very close relationship between databases and XML. It's very easy to convert between typical data and an XML file. To illustrate, I've added a button to the Access–XML form that displays the contacts database as an XML file. Figure 11.44 illustrates this function in action:

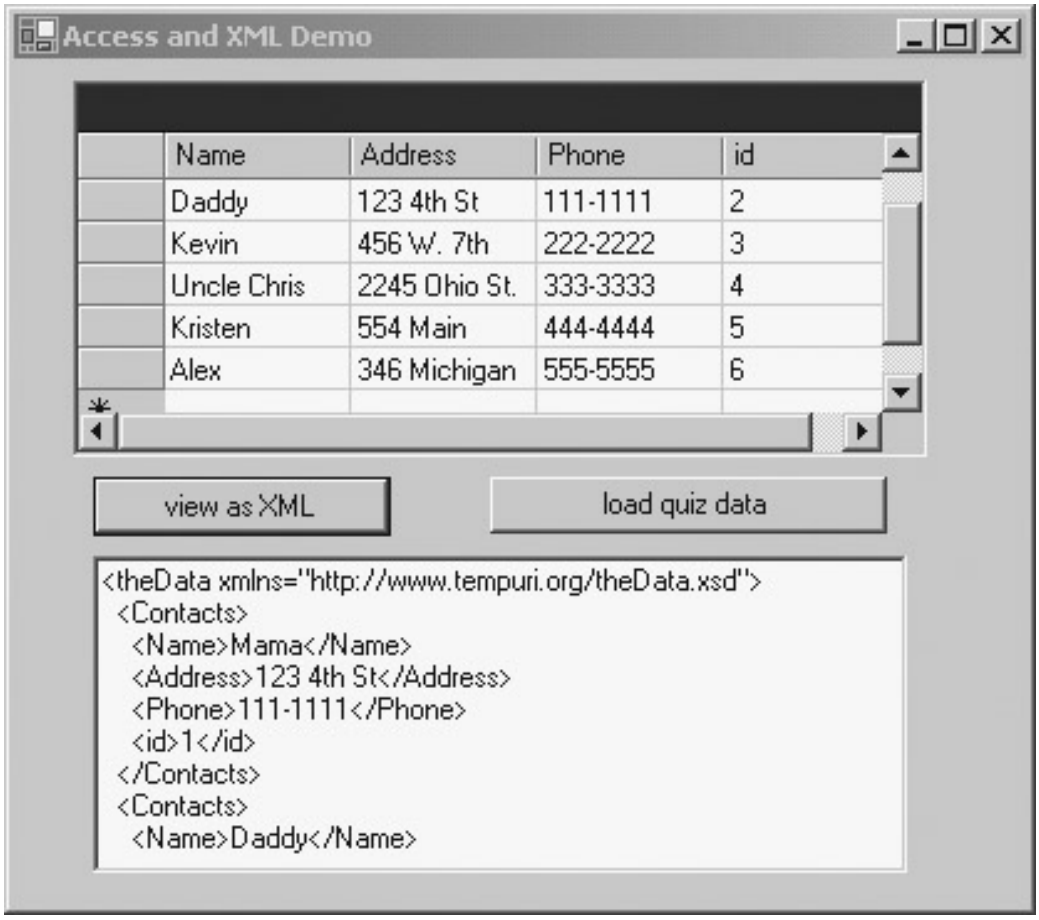


Figure 11.44: The text box displays an XML version of the Access database.
The code in the View as XML button illustrates how this is done:

```
txtXML.Text = myDS.GetXml();
```

The Dataset class has a GetXml() method which extracts XML data from a data set. The resulting string can be stored as a file or be manipulated like any other XML data.

Reading from XML to a Data Source

You also can easily read an XML file and use it as a data source. If you press the Read Quiz Data button, the program loads up the quiz XML from Chapter 10 and binds it to the data grid, as shown in Figure 11.45.

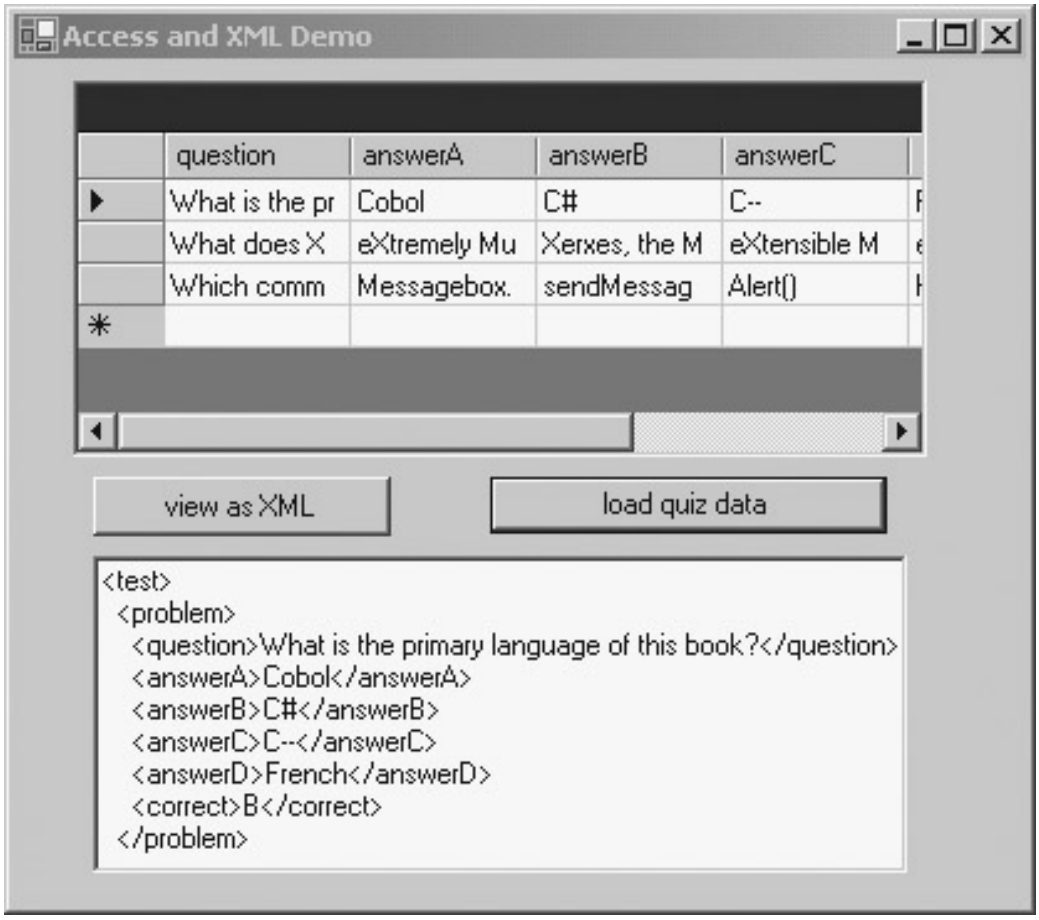


Figure 11.45: The XML quiz file from the last chapter can be viewed as a data set.

Creating the SpyMaster Database

The SpyMaster database puts together all the database concepts you have seen so far and adds a few more, to build a complete agent–handling solution. The SpyMaster program reassembles all the data taken apart in the data normalization process and puts it in a form that makes sense to the user.

Building the Main Form

Like many projects, the SpyMaster program features a main screen as a control panel. Each button on the main screen calls another editor. Because the code for the main form is so much like code you've seen before, I won't reproduce it here. You can see it on the CD–ROM if you wish.

Trick When I was debugging a particular form, I changed my main form's Main() method so it immediately called up the form I was working on. For example, when I was working on the Agent Editor form (which I worked on for quite some time, incidentally) I had the following code in the Main() method of the main form:

```
Application.Run(new AgentEdit());
```

This caused the Agent Edit form to pop up immediately without the menu screen ever appearing. Of course, when you're done debugging, you'll need to change the Main() method back so it starts itself up instead.

Some of the editing forms are much simpler than the others, so I will show them to you from simple to complex, rather than in the order the buttons appear on the main form.

Editing the Assignments

It is reasonably easy to edit any data table in your original database. The EditAssignments form enables the user to modify the Assignments table and add new assignments. The visual layout of EditAssignments is shown in Figure 11.46.

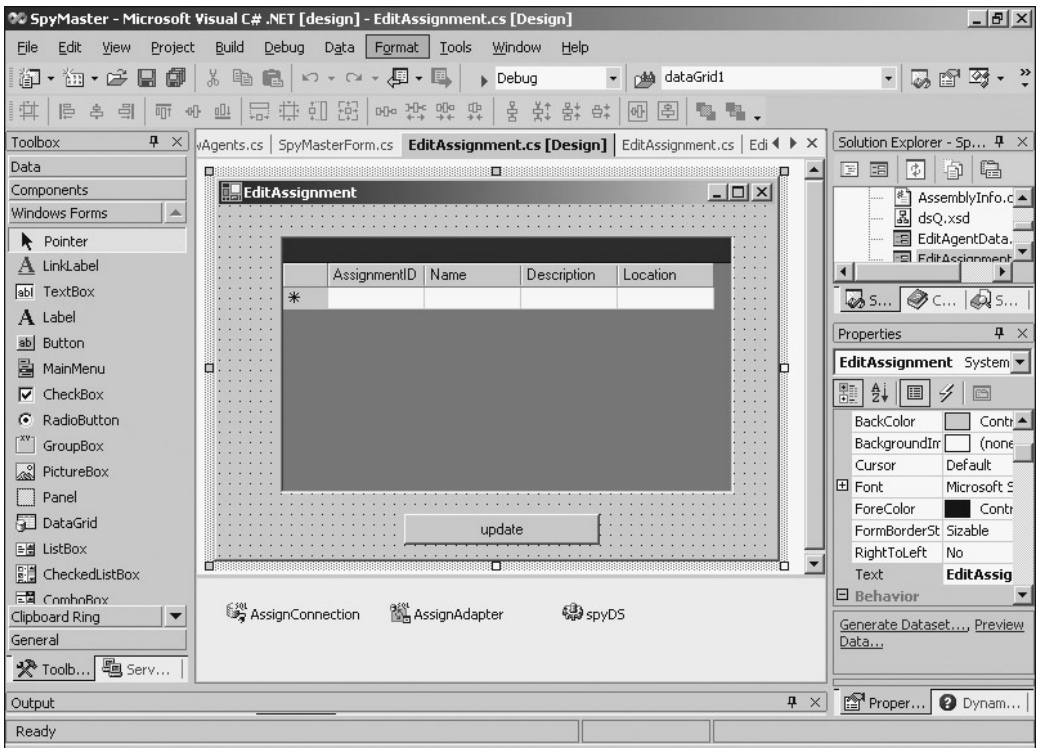


Figure 11.46: All that's needed is a data grid, a button, and some data connection controls.
I drug the Assignment table to the form and renamed my connection and adapter objects. I then created a data set based on the Adapter, renamed it, and bound the data grid to the data set.

> **Trap** If you want the editor to be able to update information, your data set must point to a table, not a view. Views cannot be edited because they often come from more than one table.

The only initialization necessary is to fill the data set from the adapter.

```
private void EditAssignment_Load(object sender,
        System.EventArgs e) {
  AssignAdapter.Fill(spyDS);
}
```

The user can modify elements in the data grid. When the user presses the Update button, the code to update the database is simplicity:

```
private void btnUpdate_Click(object sender,
System.EventArgs e) {
  AssignAdapter.Update(spyDS);
}
```

Remember that the data set is just a copy of the data. To make changes to the original database, you must use the Update() method of the appropriate data adapter. This method requires you to send a data set with the new information. Because the data set is bound to the grid, any changes made in the data grid are reflected on the data set.

Hint This is one of the advantages of .NET's data model: The user can experiment with changes to the data set without disturbing the original database. No changes are made to the original database until the adapter's Update() method is called. This gives you a chance to check the code and make sure the changes work before changing the original database.

Editing the Specialties

Editing the Specialties table is just like editing the Assignments table. The EditSpecialties form looks very similar to EditAssignments, as you can see from Figure 11.47.

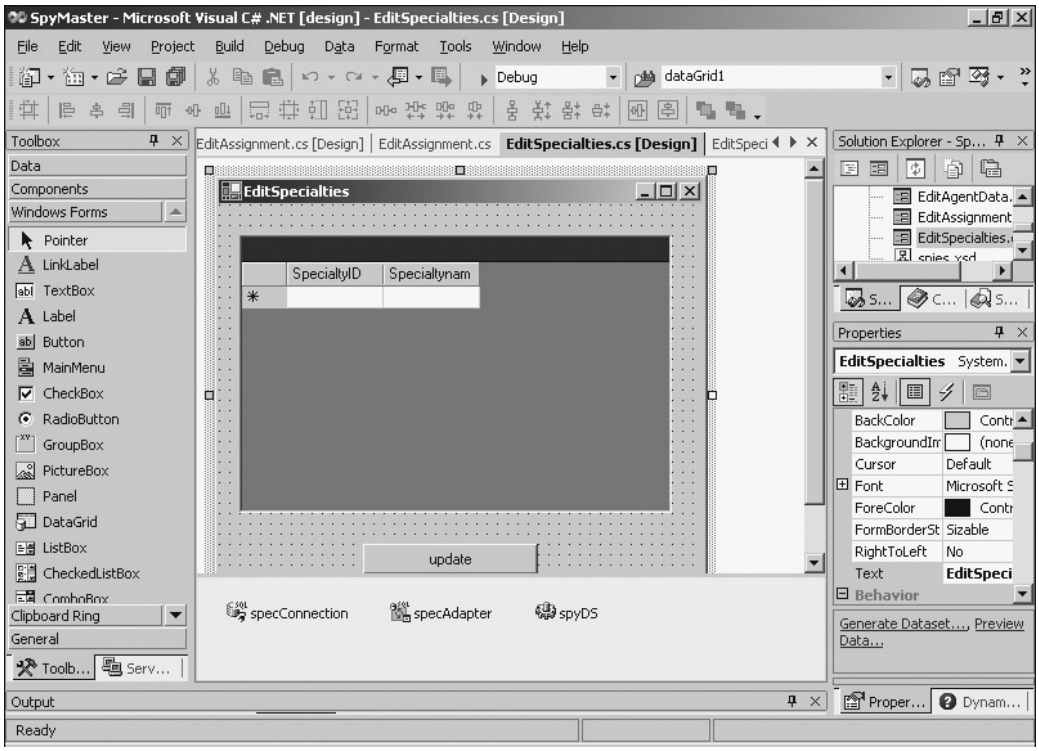


Figure 11.47: The EditSpecialties form has a data grid, a button, and data connection objects much like EditAssignments.

The form's load method fills the data table just like in EditAssignments.

```
private void button1_Click(object sender,
        System.EventArgs e) {
  specAdapter.Update(spyDS);
}
```

This form is directly connected to the Specialties table. Once again, the Update button simply calls the data adapter's Update() method.

```
private void button1_Click(object sender,
System.EventArgs e) {
  specAdapter.Update(spyDS);
}
```

Viewing the Agents

It's a little trickier to edit the agent information, because this information comes from a number of tables. However, viewing the agent information is not too tricky if you start with a data view.

Building the Visual Layout of ViewAgents

The ViewAgents form is based on a couple of data views. It is featured in Figure 11.48.

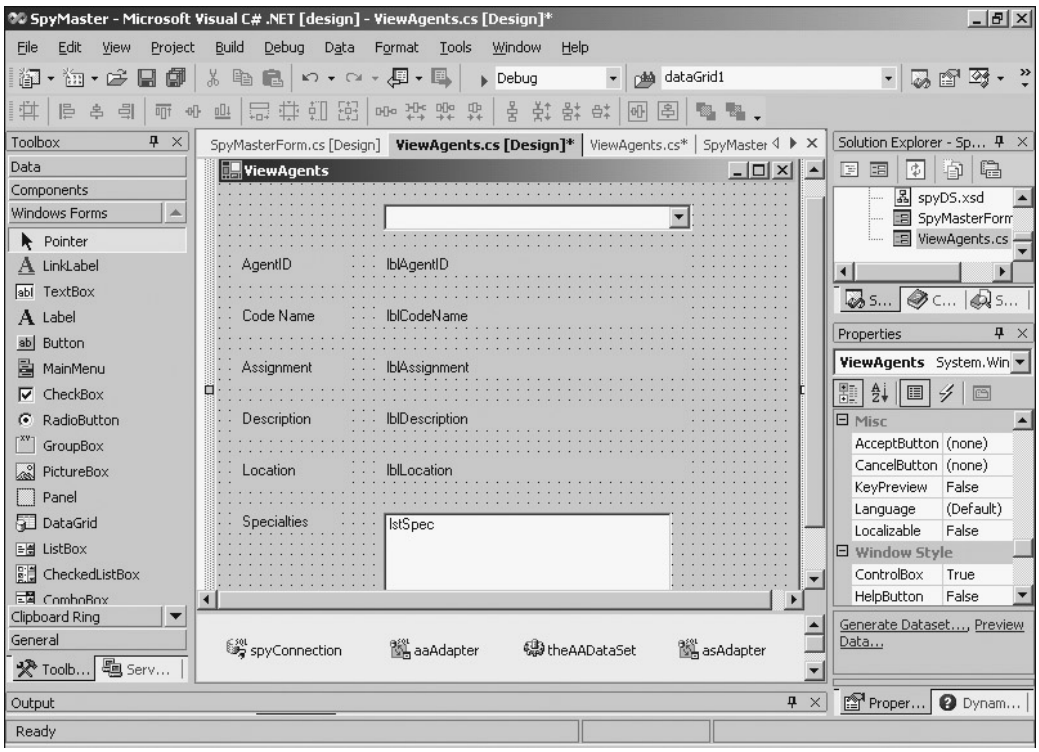


Figure 11.48: The top combo box is used to choose an agent. The current agent's data is shown on the various labels, and the list box displays all the skills associated with the current agent.

Notice that I have made multiple data adapters. The AAAdapter connects to a view called Agents_Assignments. The data set called theAAdataSet is based on

AAAdapter. I also created another adapter called ASAdapter, which connects to the Agent_Specialty view. The ASdataSet connects to the ASAdapter. Note that only one Connection object is necessary, because both adapters can use the same connection. (It's also possible to use only one data set, as I'll illustrate in the EditAgent form, but multiple data sets can be quite a bit easier to work with.)

Connecting the Combo Box

Combo boxes and list boxes can be bound to data sets just like data grids. List and combo boxes have a DataSource property that can be set to any data set. The DisplayMember property determines which field of the DataSource are displayed. I set the DataSource to theAADataSet (which is the data set corresponding to the Agents_Assignments view). I set the DisplayMember property to Agent_Assignment.CodeName. When the form is displayed, the combo box is automatically populated with all the codenames from the Agent_Assignment view. Because the data is displayed in order, the SelectedIndex property of any code name on the list also refers to the ID of the corresponding agent.

Binding the Labels to the Data Set

Most of the labels also are bound to theAADataSet. Each label has a DataSource and DataMember property which can be set to determine exactly what field of what data set is displayed. If the user chooses a new element from the combo box, all the other labels on the form are automatically updated to display the corresponding agent's information. If the components are bound to the data sets appropriately, there is no need to write any code to update the labels.

Working with Specialty Data

The specialty data does require some work, because it is not available on the Agent_Assignment view. I wrote a method called showSpecialties() (called whenever the Agents combo is changed) that re−populates the specialties list box based on a custom query.

```
private void showSpecialties(){
  //populate the specialties list box
  lstSpec.Items.Clear();
   //reset the agent-specialty query
  theASDataSet.Agent_Spec.Clear();
  asAdapter.SelectCommand.CommandText =
    "SELECT Specialtyname FROM Agent_Spec " +
    "WHERE CodeName = '" +
    lblCodeName.Text + "'";
  asAdapter.Fill(theASDataSet);

  //copy each element to list box
  foreach (DataRow myRow in theASDataSet.Agent_Spec.Rows){
    lstSpec.Items.Add(myRow["SpecialtyName"]);
  } // end foreach
} // end showSpecialties
```

The first thing the method does is clear the specialties listbox and the Agent_Spec table of theASDataSet. Then, I set a new selection command to asAdapter which selects only those skills related to the current agent, and filled the data set to account for the new query.

The foreach loop steps through each row in the resulting data set and copies the Specialtyname field of the row to the listbox.

> **Hint** To directly access a particular field, first extract a DataRow object from the data set's Rows collection. You can then use the field name as an index to the field. You can read the field values of any data row, but you can only change fields if the original data set is based on a table rather than a view.

Editing the Agent Data

The most challenging part of the program is editing the Agent information. This is difficult because the information regarding agents is spread across every table in the database.

The visual design and data connections for this form required some care. The visual design of the Edit Agent Data form is shown in Figure 11.49.

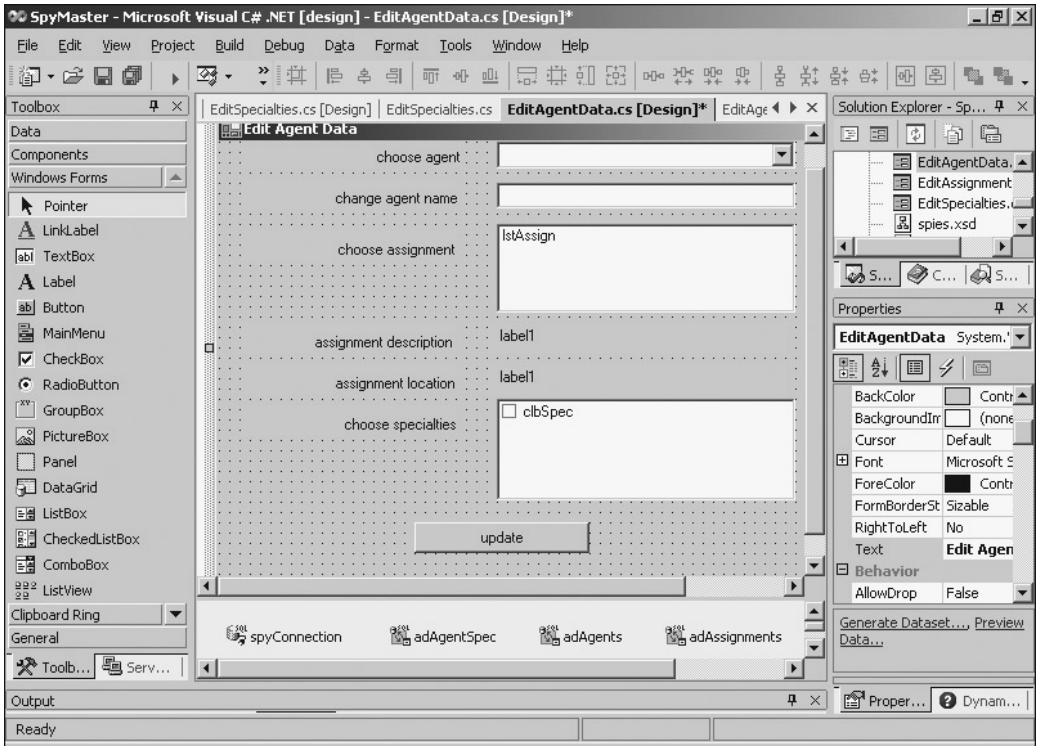


Figure 11.49: This form uses combos, list boxes, a new form of list box, and some more standard controls.

Most of the controls on the form are fairly standard, but the checked list box is new to you. This control holds a list of check boxes. It turns out to be ideal for the specialties data.

The Edit Agent Data form is different from the ViewAgent Data form in one major way. In the view form you could use a view because the information is read–only. In the edit form you must have connections to the actual table objects because the original database will be modified. Figure 11.50 illustrates the data connection structure I used for this project.

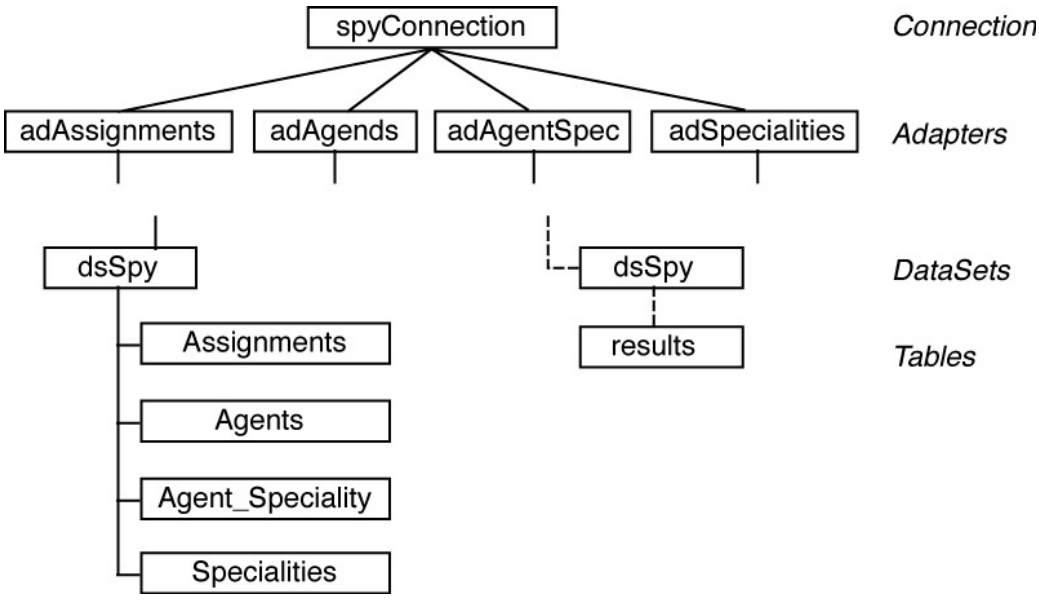


Figure 11.50: This form uses one data connection, several data adapters, and two datasets. The dotted lines indicate temporary connections.

As you can see from the diagram, one connection object attaches to the original database. I have four data adapters coming from this connection, one for each

table in the original database. Notice the naming convention I used for the data adapters. It's very important to rename data adapters because otherwise you will quickly forget which adapter is related to which table. I usually use names that indicate the table associated with the adapter.

The DataAdapter objects are used to generate two data sets. The dsSpy data set holds a copy of each table from each adapter. When you create a data set from a data adapter, you can indicate that the data set should go into an existing adapter in the Create Dataset Dialog. I used this technique to add all the tables

to the dsSpy data set. When this is done, dsSpy is a copy of all the tables in the original spy database. The other data set, called dsTemp is created in the code. It is used to generate temporary queries. It has one table called results which holds the results of any queries.

One other interesting feature of the form is a series of data grids. These grids are barely visible on the form in design time, and they are completely invisible to the user (their Visible property is set to false). These grids are each linked to one of the tables in the various data sets. I used them during debugging to make sure I knew what was going on in each table. When I was working with a particular table, I moved the corresponding grid to the center of the form to get a window on the data. Because the user does not need to see the data grids, I simply made them invisible, but kept them on the form for later debugging.

Preparing the Data Sets in the Form Load Method

The Form Load method fills the appropriate tables of the dsSpy data set from the appropriate adapters.

```
private void EditAgentData_Load(object sender,
System.EventArgs e) {
  //fill all the data adapters
  adAgentSpec.Fill(dsSpy, "Agent_Specialty");
  adAgents.Fill(dsSpy, "Agents");
  adSpecialties.Fill(dsSpy, "Specialties");
  adAssignments.Fill(dsSpy, "Assignments");
  updateForm();
}
```

Handling the Component Events

A few of the onscreen components have methods associated with them. All of these events call various custom methods. By examining the event code first, you'll have an overview of the rest of the form's methods.

```
private void cboAgent_SelectedIndexChanged(object sender,
        System.EventArgs e) {
  updateForm();
}

private void lstAssign_SelectedIndexChanged(object sender,
        System.EventArgs e) {
  getAssignInfo(lstAssign.SelectedIndex);
}

private void btnUpdate_Click(object sender, System.EventArgs e) {
  updateAgent();
  updateSpecialties();
} // end btnUpdate
```

The cboAgent combo box is bound to the CodeName field of the Agents table. Whenever the user chooses a new element from this combo box, the current agent is changed. Some elements are automatically changed, but many of them require some additional code. The updateForm() method is used to update the entire display to reflect the current agent's status.

Likewise, whenever the user chooses a new assignment, the assignment description and assignment location labels should change. The getAssignInfo()handles this duty.

When the user is ready to update an agent's information, he or she presses the btnUpdate button. The data that appears on the EditAgentData form is actually stored in two different tables, and it's necessary to update them both. The updateAgent() method updates the Agent table based on the current settings of the form, and the updateSpecialties() method updates the Agent_Specialty table based on the current settings.

Getting the Assignment Information

Most of the code in the EditAgentData form involves creating and viewing data from custom queries. The getAssignInfo() method uses this technique to figure out the values to put in the description and location labels. This method expects an assignment ID as a parameter.

```
private void getAssignInfo(int assignID){
  //use assignID to figure out description, location
  //find only the current spy's assignment
  DataSet dsTemp;
  DataRow tempRow;
  string query;

  query = "SELECT * FROM Assignments WHERE AssignmentID = ";
  query += Convert.ToString(assignID);
  adAssignments.SelectCommand.CommandText = query;
  dsTemp = new DataSet();
  adAssignments.Fill(dsTemp, "results");
  dgTemp.SetDataBinding(dsTemp, "results");

  //get a row
  tempRow = dsTemp.Tables["results"].Rows[0];
  lblDescription.Text = tempRow["Description"].ToString();
  lblLocation.Text = tempRow["Location"].ToString();
} // end getAssignInfo
```

I began by creating a temporary data set called dsTemp, a temporary DataRow object called tempRow, and a string called query. I need to know the values of the description and location fields of the Assignments table. If I'm looking for the data associated with assignment number 1, I can use the following SQL query:

```
SELECT * FROM Assignments WHERE AssignmentID = 1
```

I built a form of this statement in the query variable, but I used the value of the assignID variable. Note that although the assignID is an integer, SQL statements are strings, so I needed to convert assignID to a string.

I then reset dsTemp, and filled it from adAssignments (the adapter associated with the Assignments table). The results (which should be one row) are stored to the results table of dsTemp and this table is bound to a datagrid called dgTemp. (Recall that dgTemp is not visible in the final version of the program. It's still extremely handy because I could use it to ensure the query was working as expected before I did anything dangerous with the data.) The query should only return one row,

which is row 0. I copied that row to the tempRow variable to simplify the next couple of lines of code. Finally, I copied the description and location fields from tempRow to the text boxes.

Trap The value of a DataRow's fields are returned as an object type. You usually have to do some sort of conversion to get the data into the type you need. In this case I used the object's ToString() method to convert the field values to strings.

You see this same general strategy many times throughout this form's code. In general, it goes like this:

1. Determine the information you want.
2. Construct an SQL query to extract the information.
3. Attach that query to the appropriate data adapter.
4. Fill a temporary data set with the results.
5. Examine the rows of the data set for more details.

Updating the Form to Reflect the Current Agent

The updateForm() method is called from the form's load event, and whenever the user changes the agent in cboAgent.

```
private void updateForm(){
  fillAssignments();
  fillSpecialties();
} // end updateForm
```

It simply calls two other methods. Originally, all of the code in fillAssignments() and fillSpecialties() was in the updateForm() method, but this method quickly became unwieldy, so I split the data into smaller segments.

Filling Up the Assignments List Box

The fillAssignments() method uses a form of the algorithm described in the getAssignInfo() method.

First, the method creates a number of variables. You've already been introduced to query, dsTemp, and tempRow in the getAssignInfo() method. The agentID variable is an integer holding the ID of the current agent. I extracted this value from the SelectedIndex property of cboAgent. The first order of business is to populate the assignments list box. There is no need for a special query for this because the dsSpy.Assignments table already has all the assignments listed. I used a foreach loop to step through each row of the Assignments table. I extracted the Name field from each row and added it to the lstAssign list box.

Hint You might wonder why I didn't need to convert the name field to a string before adding it to the list box. I didn't because technically the Listbox.Items.Add() method can accept an object as a parameter, and the field is returned as an object.

Determining which assignment should be selected requires a query, because I only want to know which assignment is associated with the current agent. I built a query that will select the current agent from the Agents table. I cleared the dsTemp data set and refilled it from the appropriate adapter after applying the query. The new row appears as the results table of the dsTemp data set. I put the row into the tempRow variable and extracted the AssignmentID field from it. I then copied the value of assignID to the SelectedIndex property of lstAssign. This has the effect of highlighting whichever assignment is associated with the currently displayed spy. Because the SelectedIndex

might have changed, I called getAssignInfo() to ensure that the description and location labels were updated.

```
private void fillAssignments(){
  string query = "";
  int agentID = cboAgent.SelectedIndex;
  DataSet dsTemp = new DataSet();
  DataRow tempRow;

  //fill assignments list box
  foreach (DataRow myRow in dsSpy.Assignments.Rows){
    lstAssign.Items.Add(myRow["Name"]);
  } // end foreach

  //select appropriate assignment
  //begin by putting current agent row in dsTemp
  query = "SELECT * from Agents WHERE AgentID = " + agentID;
  adAgents.SelectCommand.CommandText = query;
  dsTemp.Clear();
  adAgents.Fill(dsTemp, "results");
  dgTemp.SetDataBinding(dsTemp, "results");

  //result is one row, grab AssignmentID
  tempRow = dsTemp.Tables["results"].Rows[0];
  int assignID = Convert.ToInt32(tempRow["AssignmentID"]);
  lstAssign.SelectedIndex = assignID;

  //fill up the assignment labels, too.
  getAssignInfo(assignID);

} // end fillAssignments
```

Trap Although this example is easy to understand, it poses a serious security threat. If a value were passed, such as DELETE FROM Agents; your table could be wiped out. A better way of writing the statement is as follows: adAgents.SelectCommand = new SqlCommand("SELECT * FROM Agents WHERE AgentID=@AgentID", new SqlConnection(connStr)); adAgents SelectCommand.Parameters.Add("@AgentID", SqlDbType.Integer); adAgents SelectCommand.Parameters["@AgentID"].Value=agentID;

Filling Up the Specialties CheckedListBox

The Specialties field enables each spy to have any number of specialties. This can be difficult to display. Although it is possible to allow multiple selections in a list box, this use of a list box is somewhat non–intuitive. C# includes a new type of list box called the CheckedListBox that is perfect for this type of situation. A checked listbox has a number of items, just like a normal list box. Each item has a check box associated with it. You can set the value of the check box for any item in the list with the SetItemChecked() method.

```
private void fillSpecialties(){
  //fill clbSpec with specialties
  string query = "";
  int agentID = cboAgent.SelectedIndex;
  DataSet dsTemp = new DataSet();
  DataRow tempRow;

  clbSpec.Items.Clear();
  foreach (DataRow myRow in dsSpy.Specialties.Rows){
    clbSpec.Items.Add(myRow["Specialtyname"]);
```

```
  } // end foreach

  //find all Agent_Spec rows for current agent
  query = "SELECT * FROM Agent_Specialty WHERE ";
  query += "AgentID = " + agentID;

  dsTemp = new DataSet();
  adAgentSpec.SelectCommand.CommandText = query;
  adAgentSpec.Fill(dsTemp, "results");
  dgTemp.SetDataBinding(dsTemp, "results");

  //preset all items in clbSpec to unchecked
  for(int i = 0; i < clbSpec.Items.Count; i++){
    clbSpec.SetItemChecked(i, false);
  } // end for loop

  //check current spy's skills in clbSpec
  foreach (DataRow myRow in dsTemp.Tables["results"].Rows){
    int specID = Convert.ToInt32(myRow["SpecialtyID"]);
    clbSpec.SetItemChecked(specID, true);
  } // end foreach

} // end fillSpecialties
```

I began by clearing clbSpec, and adding each specialty name to the checked list box. I simply stepped through all the rows in the Specialties table and added the Specialtyname field to the list box. Then I created a query that returned all the records from the Agent_Specialty table that relate to the current agent (determined by the agentID variable.) I preset each value in cblSpec to false to clear out any values that might have been there from an earlier agent. Then I looked through the query and checked the specialty associated with each record in the query.

Updating the Agent Data

After making changes, the user can choose to update the agent. This actually occurs in two distinct phases. First, it is necessary to update the Agents table. The agent's code name and assignment are stored in the Agents table, but they are not directly entered by the user.

```
private void updateAgent(){
  //updates the agents table
  DataRow agentRow;
  int agentID = cboAgent.SelectedIndex;
  int assignID = lstAssign.SelectedIndex;

  agentRow = dsSpy.Agents.Rows[agentID];

  //Change code name if new name is in text field
  if (txtCodeName.Text != ""){
    agentRow["CodeName"] = txtCodeName.Text;
    txtCodeName.Text = "";
  } // end if

  //change assignment based on lstAssign
  agentRow["AssignmentID"] = assignID;

  //update the agent in the main database
  dsSpy.AcceptChanges();
  adAgents.Update(dsSpy, "Agents");
  lstAssign.SelectedIndex = assignID;

} // end updateAgent
```

I created integers to hold the agentID and assignID. These values are easily determined by reading the SelectedIndex property of the associated list boxes. I then pulled the current agent's data row from the dsSpy.Agents table.

In the Real World

What is reusability as it applies to programming? Reusability in programming is in fact smart programming. When faced with routine programming tasks, smart programmers create reusable code through classes or functions that save them time and their employer's money.

Although often found in the object–oriented paradigm, reusability can and should be implemented in other paradigms, such as the event–driven model of Visual Basic, through functions and subprocedures.

Any time you find yourself in a situation that will or could require repeated code, go ahead and create modularized or reusable code through procedures.

I enabled the user to change the agent's name by typing in the textbox. Although I could have let the user type directly into the combo box, it turns out to be cleaner to have a text box set aside for this purpose. (In fact, I set the combo box to act like a drop–down list box so the user cannot directly type into it.) If the text box is blank (which is its default state) nothing happens. However, if there is a value in the text box, that value is copied over to the CodeName field of agentRow. I then reset the text box to be empty so the code name isn't changed for the next agent unless the user types something new in the text box.

I copied the value of the assignID variable to the AssignmentID field of agentRow.

Finally, I updated the local data set with a call to dsSpy.AcceptChanges(). This command tells the data set to register any changes made to the data. The data set is only a copy of the actual database. To make permanent changes to the original database, I used the update member of the appropriate data adapter. Remember, you can only update adapters based on data tables. As I tested this project, I discovered that sometimes the wrong element of the assignment list box is sometimes highlighted, so I reset the selectedIndex property to assignID. This ensures that the form's display is always synchronized with the data set.

Updating the Specialty Data

All the data about one agent in the Agents table resides on one record, which is easy to extract and update. The specialty data was trickier to update, because the data about one agent can span any number of records. Instead of simply modifying one existing record, I had to delete any old records for the agent and add new ones. Remember, the Agent_Specialty table works by having one record for each *relationship* between agents and specialties. It took a couple of queries to make this happen.

```
private void updateSpecialties(){
  //find all specialties associated with this agent
  try {
    string query;
    DataSet dsTemp = new DataSet();
    DataRow tempRow;
    int agentID = cboAgent.SelectedIndex;
```

```
    //find all current rows for this agent
    query = "SELECT * FROM Agent_Specialty ";
    query += "WHERE AgentID = ";
    query += agentID.ToString();

    //delete rows from database
    adAgentSpec.SelectCommand.CommandText = query;
    adAgentSpec.Fill(dsSpy, "Agent_Specialty");
    foreach (DataRow myRow in dsSpy.Agent_Specialty.Rows){
      myRow.Delete();
    } // end foreach
    //adAgentSpec.Update(dsSpy, "Agent_Specialty");

    //find the largest id
    query = "SELECT MAX(Agent_SpecialtyID) FROM Agent_Specialty";
    adAgentSpec.SelectCommand.CommandText= query;
    dsTemp = new DataSet();
    adAgentSpec.Fill(dsTemp, "results");
    tempRow = dsTemp.Tables["results"].Rows[0];
    int largestID = Convert.ToInt32(tempRow[0]);
    int newID = largestID + 1;

    //add rows
    foreach (int specID in clbSpec.CheckedIndices){
      dsSpy.Agent_Specialty.AddAgent_SpecialtyRow(
        newID, agentID, specID);
      newID++;
    } // end foreach
    dsSpy.AcceptChanges();
    adAgentSpec.Update(dsSpy, "Agent_Specialty");
    dgTemp.SetDataBinding(dsSpy, "Agent_Specialty");
  } catch (Exception exc){
    MessageBox.Show(exc.Message);
  } // end try

} // end updateSpecialties();
```

After creating the now–familiar query, DataSet, and DataTable variables, I created a query designed to return all the records of the Agent_Specialty table pertaining to the currently selected agent. I then deleted each of these rows. This is similar to clearing a list box before repopulating it. I wanted to ensure that any elements already in the database are cleared, and then add new records to handle changes in the data. Each new row requires a unique integer for its ID field. I used a special query to find the maximum value of the key field.

```
SELECT MAX(Agent_SpecialtyID) FROM Agent_Specialty
```

This query returns a special table with one row and one column. The value of this table is the largest value in the Agent_Specialty field. I then added one to that value and stored it in newID. This provides an integer guaranteed to not already be in the database.

I added a new row to the Agent_Specialty table by invoking the AddAgent_SpecialtyRow() method of the Agent_Specialty property of the dsSpy object. Recall that the Agent_Specialty property is a custom member of the extended DataSet object, and this property has a custom member to enable adding a row.

Hint You also can add a row to a table referred by a generic DataSet. Regular data set tables have a newRow property that automatically generates a new row based on the data set's schema. You then need to explicitly fill each field of the new row.

Finally, I used the acceptChanges() method of dsSpy and the Update() method of adAgentSpec to update the database with the new values.

Notice that I placed all of the primary code for the method inside a try−catch block. This is because SQL queries can cause all kinds of debugging headaches and the try−catch structure makes it much easier to determine exactly what went wrong and how to fix it.

Summary

This chapter has introduced you to the world of data design. You have learned how to use the tools integrated into Visual Studio to create a custom database. You also have learned how to create normalized databases that prevent certain kinds of errors. You know how to use the data diagram tool to create relationships between tables, and you know how to build views and queries in the visual query tool. You also have learned how to attach these databases to your programs, both by dragging elements to the form and by hand. You've learned about ADO.NET's disconnected architecture, and the objects used to access it. You built custom queries and used them to view, edit, and update data.

Challenges

- Improve the SpyMaster database so the edit assignments and edit specialties screens use text boxes rather than data grids. In particular, do not let the user enter the primary key directly, but create it automatically.
- Add a feature to the edit agents screen that enables the user to add a new agent.
- Create a relational database to manage some kind of data in your life (grades, your music collection, or whatever). Build a program to manage the data.
- Write an adventure game. Store the information about the dungeon, player, and monsters in a relational data structure.

List of Figures

Chapter 1: Basic Input and Output: A Mini Adventure

Figure 1.1: The game begins by asking the user a few questions.
Figure 1.2: The user's answers result in a silly story.
Figure 1.3: In C# programming, you have code inside methods, which are inside classes, which are inside namespaces
Figure 1.4: Here's the .NET documentation. I've expanded the tree on the left to show the various namespaces available in the .NET environment.
Figure 1.5: Some classes in the System namespace. The Console has features for communicating with the user that will be helpful.
Figure 1.6: The members of the Console class.
Figure 1.7: As advertised, the program says "Hello, World!"
Figure 1.8: The Visual Studio IDE as it appears on my computer.
Figure 1.9: The New Project dialog box is where you determine the programming language, the project's, and the type of project your are writing.
Figure 1.10: The HelloWorld program displayed in the code window.
Figure 1.11: The Play button compiles and runs your program.
Figure 1.12: A cheerful greeting from the Hello World program.
Figure 1.13: The squiggle at the end of the WriteLine() command indicates a missing semicolon.
Figure 1.14: The user types a response and receives a customized greeting.
Figure 1.15: This program demonstrates several interesting problems.

Chapter 2: Branching and Operators: The Math Game

Figure 2.1: The program asks the user simple math questions.
Figure 2.2: When the user is finished, the program reports a score.
Figure 2.3: The computer can do math, but it gave some strange results here. Is 5 divided by 4 really 1?
Figure 2.4: Although the console works only with string values, you can convert strings to whatever type of variable you wish.
Figure 2.5: The convert object can convert nearly any variable type to any other variable type.
Figure 2.6: Apparently, Bill Gates has been here!
Figure 2.7: If James Gosling (the primary developer of the Java language) shows up, the program can respond appropriately.
Figure 2.8: If users other than Bill Gates or James Gosling play this game, the program personally greets them.
Figure 2.9: The program "rolls up" two six−sided dice.

Chapter 3: Loops and Strings: The Pig Latin Program

Figure 3.1: The title of this book sounds very classy when translated into pig latin.
Figure 3.2: You can do many interesting things with string variables, including converting to upper and lower case, searching for a phrase, and determining the length of a phrase.
Figure 3.3: The Object Browser enables you to get online help on objects quickly.
Figure 3.4: The bean counter uses a for loop to repeat behavior.
Figure 3.5: The highlight indicates the current line being executed, and the Autos window describes the value of the variables.

Figure 3.6: This bean counter can count by fives, backwards, and pulls the words out of a sentence one at a time.
Figure 3.7: The troll won't let you pass until you say the magic word.
Figure 3.8: The first time through the loop, the condition is false.
Figure 3.9: Now response is equal to the Answer, so the condition is false.

Chapter 4: Objects and Encapsulation: The Critter Program

Figure 4.1: Introducing your critter. Go ahead and talk to it!
Figure 4.2: The critter tells you about itself.
Figure 4.3: After talking for a while, the critter becomes melancholy.
Figure 4.4: Without appropriate attention, the critter becomes angry.
Figure 4.5: The cost of neglect can be a sad and lifeless critter. Fortunately, with enough food and love, it can be revived.
Figure 4.6: The Main() method features a few commands. You can probably guess what each one does.
Figure 4.7: The Song program re–creates the song This Old Man.
Figure 4.8: The menu for the Critter program is a standard console menu.
Figure 4.9: This error message makes my program seem very unfriendly.
Figure 4.10: This version of the Critter program features a critter that knows its name.
Figure 4.11: You can change the name property so that the critter will reinforce special rules.

Chapter 5: Constructors, Inheritance, and Polymorphism: The Snowball Fight

Figure 5.1: Begin by entering the player names, which are important for the play–by–play description of the game.
Figure 5.2: Players have a limited number of snowballs and are more likely to hit their target when they are close to it.
Figure 5.3: With a good strategy, you can beat the computer much of the time, but not always.
Figure 5.4: The critter viewer demonstrates the basic functionality of the critter.
Figure 5.5: To create a new class, choose Add Class from the Project menu, then click Class.
Figure 5.6: The Class View is used to navigate the entire project.
Figure 5.7: Although not apparent from the output, each critter uses a different constructor. The last one begins with a blank name because it was called with no parameters.
Figure 5.8: The output looks very familiar, but Dolly is actually a clone!
Figure 5.9: The glitter critter is like normal critters, but it has a shine() method.

Chapter 6: Creating a Windows Program: The Visual Critter

Figure 6.1: This critter has a picture and some Windows–style controls.
Figure 6.2: When the user clicks the image, the critter speaks.
Figure 6.3: The user can rename the critter by typing in the little box.
Figure 6.4: After the user clicks the Change My Name button, the critter reports its new name. Any subsequent clicks on the critter will return the new name.
Figure 6.5: When the user chooses a new mood from the drop–down list, the image changes accordingly.
Figure 6.6: The elevator shaft is near the top, and the critter image is very small.
Figure 6.7: When the user moves the scroll bar down, the critter grows.
Figure 6.8: You can choose to make a Windows project when you start a new C# project.
Figure 6.9: When you build a Windows project, the IDE changes to add graphical features.

Figure 6.10: The program says a Windows−style hi, without a line of code!
Figure 6.11: This code creates the Hello GUI program.
Figure 6.12: The Form class is a very powerful part of the System. Windows.Forms namespace, with many useful characteristics.
Figure 6.13: The temptation is almost irresistible.
Figure 6.14: When the user (inevitably) clicks the button, the program responds to the event.
Figure 6.15: Double−clicking any event automatically creates an event handler for that event.
Figure 6.16: The user has used familiar interface tools to specify a 10−point Arial font with italic style.
Figure 6.17: The user has selected a different font. Notice that the font can be both bold and italic but can have only one size.
Figure 6.18: Sketch out the types and names of objects in your form.
Figure 6.19: The Font class has a three−parameter constructor that builds a font based on exactly the information the form generates.
Figure 6.20: In its default state, the picture is small, and the scroll bar is all the way to the left.
Figure 6.21: When the user drags the scroll bar, the image becomes larger.
Figure 6.22: My initial sketch of the Sizer program.
Figure 6.23: All the picture boxes are the same, except for the setting of SizeMode.
Figure 6.24: This sketch helped me to define how my program would be built.
Figure 6.25: The form is easy to build if you have a sketch to follow.

Chapter 7: Timers and Animation: The Lunar Lander

Figure 7.1: The landing craft starts with a random speed and direction—and limited fuel.
Figure 7.2: When the user presses lowercase a, both events display the value a, but they disagree on capitalization.
Figure 7.3: If the user presses a control key, only the KeyDown event can interpret the value.
Figure 7.4: The KeyDown event can also respond to arrow keys, function keys, and other special keyboard inputs.
Figure 7.5: The KeyPressEventArgs object has properties and methods describing which key was pressed.
Figure 7.6: The KeyEventArgs object has a longer list of features than KeyPressedEvents.
Figure 7.7: Each time the user clicks the Next button, the image changes. The result is the illusion of a spinning globe.
Figure 7.8: The ImageList control resides in a special place outside the main screen.
Figure 7.9: When you modify the Images property of an ImageList, the program provides a special editor where you can select each image from your file system.
Figure 7.10: You can’t see it here, but the image changes 10 times per second.
Figure 7.11: The Auto Spin program features two invisible controls and one picture box that is apparent to the user.
Figure 7.12: When the program starts, the arrow is moving to the right.
Figure 7.13: When the user presses an arrow key, the arrow image changes and moves in the indicated direction.
Figure 7.14: The ball is moving towards the target, and the background is white.
Figure 7.15: When the ball moves over the target, the form’s background color changes to black.
Figure 7.16: Nothing happens until the user clicks the button.
Figure 7.17: Notice the question icon, the two buttons, and the labels on the buttons.
Figure 7.18: Apparently, there is a mechanism for reading which button was clicked.

Chapter 8: Arrays: The Soccer Game

Figure 8.1: The player within the square outline has the ball. He can either pass to another player or make a shot on the goal.
Figure 8.2: A golf scorecard is a good example of an array in everyday life.
Figure 8.3: The label says zero before the user presses any buttons.
Figure 8.4: After the user presses the button, the value changes.
Figure 8.5: An array of programming language names is displayed in a list box. The array is not in any particular order.
Figure 8.6: When the user chooses the sort option, the array appears in alphabetical order.
Figure 8.7: The methods of the Array object make it easy to work with, especially if you are working with some kind of array.
Figure 8.8: The user can search for an element to see where it is in the array.
Figure 8.9: When the user clicks the forEach button, the elements of the array are shown one at a time.
Figure 8.10: The halfback should complete a pass to the wing 80 percent of the time.
Figure 8.11: The drop−down menu is automatically populated with all methods detected by the IDE.
Figure 8.12: The form has a button and responds to the button's click event.
Figure 8.13: The Visual Designer shows no components at all!
Figure 8.14: The images stored in the image list control.

Chapter 9: File Handling: The Adventure Kit

Figure 9.1: The main screen provides very basic access to the other parts of the program.
Figure 9.2: The file dialog looks very familiar to experienced users of Windows.
Figure 9.3: Here's the starting situation. Try clicking Sub Deck. (The Enigma device was carried on submarines.)
Figure 9.4: The screen changes to reflect the new situation. It's not looking good, but there are plenty of places to jump.
Figure 9.5: Oh, no!
Figure 9.6: In the Dungeon Editor, you can change the values of the screen elements.
Figure 9.7: The File IO program has a large text box and buttons for saving and loading the file.
Figure 9.8: The System.IO namespace includes classes for input and output streams.
Figure 9.9: The form has one main menu, which has menu items. The menu items have submenus.
Figure 9.10: After the user makes a selection, the program prints out the number name.
Figure 9.11: The MainMenu object is below the form, and the first MenuItem object has been placed at the top of the window. (It says Type Here.)
Figure 9.12: When you start to type in a menu item, two new potential menu items appear!
Figure 9.13: Every element opens up the possibility for two more elements.
Figure 9.14: When the user chooses Open from the File menu, a familiar−looking dialog appears.
Figure 9.15: The font dialog allows the user to set the font, including typeface, size, and style.
Figure 9.16: The user can independently set the foreground and background colors of the text editor.
Figure 9.17: The dialogs are added to the bottom of the form. Notice the menu structure as well.
Figure 9.18: The user can enter data into text boxes.
Figure 9.19: I drew a very simple dungeon to get a sense of what kind of data an adventure

game requires.
Figure 9.20: The game window features several labels and a menu.
Figure 9.21: The layout for the editor is similar to the game form, but the controls can be edited.

Chapter 10: Chapter Basic XML: The Quiz Maker

Figure 10.1: The main screen allows quick access to the other forms of the quiz program.
Figure 10.2: The quiz interface is quite simple.
Figure 10.3: The user can enter quiz data using an editor, much like the Adventure Kit game.
Figure 10.4: The test contains problems, which can contain a question, answers, and the correct answer.
Figure 10.5: The Visual Studio XML editor automatically indents your code and creates a closing tag for each opening tag.
Figure 10.6: After you define your data set, you can enter it in, just like a database.
Figure 10.7: The starting node is named xml, but it isn't very interesting.
Figure 10.8: The test node has a lot more information. (Almost too much!)
Figure 10.9: I have clicked the first problem, and the viewer is now looking at the problem node.
Figure 10.10: Now the viewer is pointed at the last child of the test node.
Figure 10.11: You might be surprised that there still isn't anything useful in the value field.
Figure 10.12: This diagram more accurately reflects how .NET sees an XML document.
Figure 10.13: Most of the visual interface is composed of labels, with a few buttons and a list box added for navigation.
Figure 10.14: The XML displayed in the right panel was created by the program.
Figure 10.15: After entering a new element in the text boxes, pressing the Add button, and displaying the XML, a new element is visible in the code.
Figure 10.16: The document contains a contact, which is a group of person elements. Each person consists of a name, address, and phone number.
Figure 10.17: The native XML code is complete but very difficult for humans to read.
Figure 10.18: As usual, the layout of the main form is extremely simple.
Figure 10.19: The FrmQuiz form has a label for the question and radio buttons for the answers
Figure 10.20: The editor form relies on text boxes to display and retrieve the problems.

Chapter 11: Databases and ADO.NET: The Spy Database

Figure 11.1: From this main screen you can achieve world dominance (or protect it from evil).
Figure 11.2: All the information regarding a spy is available on this form.
Figure 11.3: The edit agents screen is similar to the view agents screen, except that now you can change some of the data.
Figure 11.4: Here the user can add and modify assignments in a grid.
Figure 11.5: The user can add and modify specialties. You never know when explosives and doilies will be needed on a mission.
Figure 11.6: The hierarchy tree on the left–hand side shows my machine with my machine listed as an SQL Server. The SimpleSpy database is listed as an element of my SQL Server.
Figure 11.7: A table editor reminiscent of Microsoft Access pops up.
Figure 11.8: Each agent has a code name, an assignment, and a specialty field.
Figure 11.9: The AgentID field is now the primary key of the Agents table.
Figure 11.10: Agents table after I've entered a few of my agents. (This page will self–destruct in 5 seconds…)

Figure 11.11: When you drag a table to your form, two new objects are created in the non−visible segment of your form.
Figure 11.12: The Preview Data dialog illustrates how the data looks to the program.
Figure 11.13: After pressing the Fill DataSet button, the data set appears on the dialog.
Figure 11.14: The Generate DataSet Dialog prompts you to name your new DataSet.
Figure 11.15: Setting the data DataMember property only works after you've set the DataSource property.
Figure 11.16: The program is almost ready, but the actual data has not yet been passed to the data set from the original database.
Figure 11.17: Although there was a lot of mouse jockeying involved, the database can be displayed on a form with only one line of code.
Figure 11.18: The default version shows the entire database of spies.
Figure 11.19: Now the program only shows the spy codenames.
Figure 11.20: This query returns a list of the codenames and specialties.
Figure 11.21: The query returns all the fields, but only the records of the spies who specialize in explosives.
Figure 11.22: This query limits both the number rows and the number of columns.
Figure 11.23: The query demo has been attached to the SimpleSpy database.
Figure 11.24: The form is just like QueryDemo, but it adds some other controls for building the query.
Figure 11.25: If the user types a value into the textbox and chooses a field from the listbox, only records that satisfy the resulting condition are displayed.
Figure 11.26: The Visual Query form features checkboxes, list boxes, and a couple of labels.
Figure 11.27: This version of the spy database has more information, but it also introduces a number of problems.
Figure 11.28: The Agents table is quite a bit simpler than it was before.
Figure 11.29: The Assignments table describes all the information related to a specific operation.
Figure 11.30: The data diagram tool shows the tables in your database.
Figure 11.31: You can leave the default values of the Create Relationship dialog.
Figure 11.32: The solid line indicates the relationship between the Agents and Assignments tables.
Figure 11.33: The view editor features a form of the data diagram. The relationship has been preserved.
Figure 11.34: As you can see from the bottom of the screen, this view recombines the Agents and Assignments tables.
Figure 11.35: Once you've created a view, you can attach a grid to the view as if it were a table.
Figure 11.36: The Specialties table simply lists all the various specialties.
Figure 11.37: The Agent_Specialty table serves as a bridge between the Agents table and the Specialties Table.
Figure 11.38: The Agents table connects directly with the Assignments table, but uses the Agents_Specialty table as a bridge to the Specialty table.
Figure 11.39: The Data Link Properties Dialog lets you choose from a number of different data sources.
Figure 11.40: You can now select an access database from your file system.
Figure 11.41: The toolbox (where you normally choose components such as textboxes and labels) has a data tab that provides access to several data components.
Figure 11.42: All the connections established on the current machine are available from the drop−down list.
Figure 11.43: You are prompted for a SELECT statement to initialize the data adapter.
Figure 11.44: The text box displays an XML version of the Access database.

Figure 11.45: The XML quiz file from the last chapter can be viewed as a data set.
Figure 11.46: All that's needed is a data grid, a button, and some data connection controls.
Figure 11.47: The EditSpecialties form has a data grid, a button, and data connection objects much like EditAssignments.
Figure 11.48: The top combo box is used to choose an agent. The current agent's data is shown on the various labels, and the list box displays all the skills associated with the current agent.
Figure 11.49: This form uses combos, list boxes, a new form of list box, and some more standard controls.
Figure 11.50: This form uses one data connection, several data adapters, and two datasets. The dotted lines indicate temporary connections.

List of Tables

Chapter 2: Branching and Operators: The Math Game

Table 2.1: Selected Data Types
Table 2.2: Comparison Operators

Chapter 8: Arrays: The Soccer Game

Table 8.1: The Likelihood of Success from the Fullback Spot
Table 8.2: Percentages For All Plays

Chapter 9: File Handling: The Adventure Kit

Table 9.1: Significant Classes in the System.IO Namespace
Table 9.2: A Simple Dungeon

Chapter 10: Chapter Basic XML: The Quiz Maker

Table 10.1: Selected Members of the XmlNode Class
Table 10.2: Selected Members of the XmlDocument Class

List of Sidebars

Chapter 1: Basic Input and Output: A Mini Adventure

In the Real World
Challenges

Chapter 2: Branching and Operators: The Math Game

In the Real World
In the Real World
In the Real World
In the Real World
Challenges

Chapter 3: Loops and Strings: The Pig Latin Program

In the Real World
In the Real World
In the Real World
In the Real World
In the Real World
In the Real World
Challenges

Chapter 4: Objects and Encapsulation: The Critter Program

In the Real World
In the Real World
Challenges

Chapter 5: Constructors, Inheritance, and Polymorphism: The Snowball Fight

In the Real World
In the Real World
In the Real World
Challenges

Chapter 6: Creating a Windows Program: The Visual Critter

In the Real World
In the Real World
In the Real World
1In the Real World
In the Real World
In the Real World
In the Real World
In the Real World
In the Real World
In the Real World
Challenges

Chapter 7: Timers and Animation: The Lunar Lander

In the Real World
In the Real World
In the Real World
In the Real World
In the Real World
In the Real World
Challenges

Chapter 8: Arrays: The Soccer Game

In the Real World
In the Real World
Challenges

Chapter 9: File Handling: The Adventure Kit

In the Real World
In the Real World
In the Real World
In the Real World
In the Real World
In the Real World
In the Real World
Challenges

Chapter 10: Chapter Basic XML: The Quiz Maker

In the Real World
In the Real World
In the Real World
In the Real World
In the Real World
Challenges

Chapter 11: Databases and ADO.NET: The Spy Database

A Note about the CD−ROM
Keeping Track of Your Data Structure
How Do Data Connections, Data Adapters, and Data Sets Fit Together?
Displaying the Query
In the Real World
Challenges